LŐCS GYULA

# ALGOL

# 60

## PROGRAMOZÁSI NYELV

LŐCS GYULA

# AZ ALGOL 60 PROGRAMOZÁSI NYELV

*Hatodik kiadás*

MŰSZAKI KÖNYVKIADÓ,
BUDAPEST
1978

Lektorálta:

BÉKÉSSY ANDRÁS
a matematikai tudományok doktora

© Lőcs Gyula, 1968

ETO: 681.3.06

ISBN: 963 10 2158 0

Felelős kiadó: Solt Sándor igazgató
Felelős szerkesztő: Dr. Vas Györgyné okl. alk. matematikus

# ELŐSZÓ

Az elektronikus számológépek világméretű elterjedése mindenütt — így Magyarországon is — szükségszerűen megnövelte azoknak a szakembereknek a számát, akik a számológépek programozása iránt érdeklődnek. Ez a mű bevezetést nyújt a nemzetközileg elfogadott ALGOL 60 számológép-programozási rendszerbe, mégpedig olyan olvasók számára, akik számológép-programozással eddig nem, vagy csak keveset foglalkoztak.

A könyv bevezető jellege miatt alig használtam fel elektronikus számológépekre vonatkozó olyan ismereteket, amelyeket előzőleg — legalábbis nagy vonalakban — ne világítottam volna meg. Mindössze néhány alapfogalom előzetes ismeretét tételeztem fel, olyanokét, amelyek magyarázata az ismeretterjesztő színvonalú irodalomban megtalálható; ilyenek pl. az elektronikus számológépek alapvető egységeinek elnevezése, funkciója és egymással való kapcsolata.* Könyvem első fejezete általános bevezető, amely a későbbiekben előforduló fogalmak magyarázatát tartalmazza.

A számológép-programozás gyakorlati tudomány. Elsajátítása konkrét feladatok programozása nélkül — kizárólag elméleti alapon — nem lehetséges. Éppen ezért, a kidolgozott programozási példákon kívül, a második fejezettől kezdve minden fejezet végén néhány gyakorlatot közöltem, amelyek önálló megoldása hozzájárul a programozásban való jártasság kifejlődéséhez. A könyv végén összegyűjtve található meg ezeknek a gyakorlatoknak egy-egy

* Általános bevezetőt nyújt az elektronikus számológépek elméletébe pl. [1]. [2].

lehetséges megoldása. Hangsúlyozni szeretném a „lehetséges" jelzőt. Egy programozási feladatnak ugyanis általában végtelen sok különféle megoldása van, közülük egy sereg egyenértékű, és a „legjobb" – ami egyébként is relatív fogalom – kiválasztása nem mindig lehetséges. A közölt megoldásokról sem kívánom állítani, hogy bármilyen szempontból is „legjobbak" lennének; gondolatmenetük azonban illeszkedik a megfelelő fejezet tárgyalásmódjához. Javaslom az Olvasónak, hogy akár megoldotta a feladatot, akár nem, a megoldást mindenképpen tanulmányozza át, mert a közölt változathoz fűzött megjegyzés esetleg hasznára lehet.

Mind a tárgyalás során bemutatott példák, mind a gyakorlatok anyaga főként a numerikus analízis területéről származik. Szeretném ezzel kapcsolatban megjegyezni, hogy a közölt numerikus eljárások általában olyan algoritmusok, melyeknél sokkal hatékonyabb, gyorsabb és gazdaságosabb módszerek is ismeretesek az irodalomban. Kiválasztásuk elsődleges szempontja a kényelmes – és nem utolsó sorban a rövid – programozhatóság volt. A számítási eljárások pontossága, gyorsasága, kerekítési hibákkal szemben tanúsított viselkedése stb. mind olyan kérdések, amelyek nagyon is lényegesek konkrét feladatok gépi megoldása során, a könyv célját és terjedelmét szem előtt tartva azonban ezek taglalására általában nem térhettem ki. A közölt algoritmusok általában elemieknek tekinthetők, vizsgálatuk a numerikus analizis bevezető tankönyveiben található meg. Bármely konkrét feladat elektronikus számológépen való megoldása előtt feltétlenül ajánlatos áttanulmányozni az illető szakterület számítási eljárásainak irodalmát.

Ezúton szeretnék köszönetet mondani *Békéssy András* lektornak értékes és alapos elemzéséért, amellyel segítséget nyújtott munkámhoz, továbbá mindazoknak, akik elősegítették e könyv megjelenését.

*A szerző*

*Első fejezet*

# AZ ELEKTRONIKUS SZÁMOLÓGÉPEK PROGRAMOZÁSÁNAK AUTOMATIZÁLÁSA ÉS AZ ALGOL 60 PROGRAMOZÁSI NYELV

## 1. Bevezetés

Az elmúlt másfél évtized folyamán a programmal vezérelt elektronikus digitális számológépek a tudomány, a termelés és a közgazdaság nélkülözhetetlen segédeszközeivé váltak. Alkalmazási területük rohamosan bővül, és ezzel együtt nő azoknak a szakembereknek a száma is, akik mindennapi munkájuk során kapcsolatba kerülnek ezekkel a berendezésekkel. Az elektronikus számológép egyre inkább olyan általános tudományos eszköz lesz, mint régebben az asztali számológép vagy a logarléc volt.

Magától értetődően az elektronikus számológépek széles körű elterjedésének alapvető feltétele, hogy felhasználásuk módja és szabályai könnyen elsajátíthatók legyenek, és hogy a megoldásra váró problémákat minél egyszerűbben lehessen a géppel közölni. Az elektronikus számológépek műszaki-technikai fejlesztésével párhuzamosan, a kutatók az utóbbi években igen sokat fáradoztak e követelmény minél messzebbmenő kielégítésén.

Mint ismeretes, még a legkorszerűbb elektronikus számológépek is csak viszonylag kevés elemi műveletet képesek végezni (elemi aritmetikai műveletek, adattovábbítás a gép különböző egységei és regiszterei között, egyszerű logikai döntések és operációk stb.), és a gép által megoldandó minden feladatot ezen elemi műveletek sorozatára kell elbontani. A rendelkezésre álló elemi műveletek összessége — ennélfogva a lebontás eredményeként kapott ún. *gépi program* is — géptípusonként más és más. A lebontás munkája, az ún. *kódolás* végezhető „kézileg". Ez azt jelenti, hogy a konkrét gép műveletrendszerét és a feladat jellegét ismerő személy mindenféle gépi segédeszköz nélkül dolgozza

ki a gépi programot. A kézi kódolás általában igen hosszadalmas, és nagyon sok hibaforrást tartalmaz. Külön hátránya még, hogy az egy feladatra kidolgozott program a későbbiekben is csak ugyanazon géptípusra használható fel.

A kódolás másik módja, amikor a konkrét feladat lebontását a gépi műveletek sorozatára magával a számológéppel végeztetjük el. Ennek feltétele, hogy a feladat olyan formában — olyan programozási „nyelven" — legyen megadva, amin a gép „ért", vagyis a feladatnak eleget kell tennie bizonyos formai és értelmezési szabályoknak.

Célszerűségi szempontok természetesen azt is megkívánják, hogy a szokásos matematikai jelölésekkel felírt feladat minél egyszerűbben legyen átírható az adott programozási nyelvre.

A jelen könyv bevezető az ALGOL 60 programozási nyelvbe. Ahhoz, hogy a gép egy bizonyos programozási nyelven „értsen", az szükséges, hogy rendelkezésre álljon egy speciális gépi program, ún. *fordítóprogram*, amelynek segítségével a számológép az adott programozási nyelven írt programot a gép nyelvére „lefordítja".

A programozási nyelvnek és gépi kódolás alkalmazásának nagy előnye, hogy ha egyszer egy feladat programját valamelyik programozási nyelven elkészítették, akkor ezt fel tudja használni bármely olyan gép, amely fel van szerelve a megfelelő fordítóprogrammal.

A felhasználó szempontjából a fordítóprogram a géppel együtt mindig rendelkezésre áll, és azzal bizonyos értelemben egy egységet képez. Szerkezete, működése, organizációja a felhasználót általában kevéssé érinti, ezért ezeket a kérdéseket csak annyiban tárgyaljuk, amennyiben az anyag megértéséhez szükséges. Fordítóprogramok szerkesztésének elméletével és ezzel rokon kérdések tanulmányozásával a matematikának egy modern ága, a programozáselmélet foglalkozik.*

Mielőtt rátérnénk az ALGOL 60 érdemi tárgyalására, tekintsük át nagy vonásokban a számítási feladatok gépi

* ALGOL fordítóprogramok megszerkesztésének elméleti kérdéseivel foglalkozik pl. [21], [22].

megoldásának általános menetét. Egy számítási feladat gépi megoldása a következő lépésekben megy végbe:

1. *Matematikai modellalkotás.* Ezen a feladat matematikai formába való öltöztetését, matematikai leírását értjük. A modell lehet pl. differenciálegyenlet, algebrai egyenlet vagy egyenletrendszer, valamely képlet (formula), esetleg valamiféle logikai döntéseken alapuló különböző műveletek egymásutánja stb.

2. *Algoritmizálás.* Az algoritmizálás azoknak a számítási lépéseknek a meghatározását jelenti, amelyek segítségével, a matematikai modellnek és a kiindulóadatoknak a birtokában, el lehet jutni a feladat megoldásához. A szükséges számítási lépések összességét a feladatmegoldás algoritmusának nevezzük. Tipikus matematikai modellekhez megfelelő algoritmusok kidolgozásával a numerikus analízis, az algoritmusok elvi vizsgálatával az algoritmuselmélet foglalkozik.

3. *Kódolás.* Ebben a lépésben a feladat algoritmusát a gép számára végrehajtható elemi műveletek egymásutánjára bontjuk le. A kódolásban igénybe vehetjük a gép segítségét, ha rendelkezik alkalmas fordítóprogrammal.

4. *Ellenőrzés.* Ebben a lépésben az előző három pontban leírtak ellenőrzésére kerül sor.

5. *Gépi lefuttatás.* Ez a lépés a tulajdonképpeni feladatmegoldás, amikor is az elektronikus számológép végrehajtja a matematikai modell végigszámolására kidolgozott, alkalmas módon kódolt és ellenőrzött algoritmust. Az első négy pontban leírtak befejezése után kapott végleges számológép-program lefuttatása többször is megismételhető (pl. különböző bemenő adatokkal).

Különböző fordítóprogramok segítségével könnyebbé, kényelmesebbé és hibamentesebbé tehető a kódolás, azonban semmiképpen sem válnak feleslegessé a fenti felsorolás egyéb pontjaiban leírtak, különösen a matematikai modellalkotás és az algoritmizálás. E műveletek a számológéptől — legalábbis elvben — függetlenek.

## 2. Az ALGOL 60 nyelv célja, szintjei

Egy feladatról, amely eleget tesz valamely fordítóprogram által megkövetelt formai és értelmezési szabályoknak, azt mondjuk, hogy az illető fordítóprogramnak megfelelő *forrásnyelven* vagy *bemeneti nyelven* van írva. Megemlítjük, hogy mindig a szabályokat alkotják meg elsődlegesen, és ezekhez illesztik hozzá a fordítóprogramot.

A számológép-programozás gyakorlatában jelenleg három egyezményes forrásnyelv van széles körben elterjedve: a FORTRAN, az ALGOL 60 és a COBOL.*

Az ALGOL betűszó, az ALGOrithmic Language (algoritmikus nyelv) elnevezés rövidítése. A rá vonatkozó formális és értelmezési szabályokat 1955-ben kezdték kidolgozni. A nyelv első, nemzetközileg még el nem fogadott változatát 1958-ban, végleges definícióját 1960-ban tették közzé [14], [15], [10], [12]. 1963-ban publikálták a nyelvnek egy módosított definícióját ([11], [13]), amely az időközben kiderült kisebb hiányosságok és szövegezésbeli pontatlanságok kiküszöbölése utáni, jelenleg is érvényben levő változatot közli. A 60-as szám az elnevezésben az elfogadás évére utal; annak ellenére, hogy ez idő szerint az 1963-ban elfogadott módosított verzió van érvényben, azt nem változtatták meg, mivel e módosítások valóban lényegtelenek, és a nyelv jellegét érdemben nem érintik. Az ALGOL 60 jelenleg érvényes változatát *hivatkozási nyelvnek* nevezzük. A továbbiakban, valahányszor az ALGOL 60 nyelvről beszélünk, mindig a hivatkozási nyelvre kell gondolni, hacsak ennek ellenkezőjét külön nem hangsúlyozzuk.

A hivatkozási nyelv konkrét realizációit egyes számológépeken *gépi reprezentánsoknak* nevezzük. A gépi reprezentánsok kisebb-nagyobb eltéréseket mutatnak a hivatkozási nyelvhez képest. Az eltéréseket az illető gép fordítóprogramjához mellékelt kézikönyvek sorolják fel.

Az ALGOL 60 programozási nyelv használható számítási eljárások, számológép-programok irodalmi publikálá-

* A FORTRAN nyelvre nézve lásd pl. [8], [24], a COBOL-t illetően pedig [9].

sára is. Ilyen esetben megengedhető az ALGOL-ban írt „szöveg" tipográfiai célszerűség miatti átrendezése, a hivatkozási nyelvben nem szereplő betűtípusok (pl. görög betűk) alkalmazása, továbbá a hivatkozási nyelv bizonyos jeleinek helyettesítése más jelekkel. Az ALGOL nyelv ilyen változatait *publikációs nyelveknek* nevezzük. Az ALGOL nyelv szerzői azt követelik meg a publikációs nyelvek alkalmazóitól, hogy a publikációs nyelvek jelöléseiken kívül semmiben ne különbözzenek a hivatkozási nyelvtől, és hogy a publikációs nyelvekről világos áttérési szabály legyen megadható a hivatkozási nyelvre.

A hivatkozási, a publikációs és a gépi reprezentációs nyelvek az ALGOL három szintjét alkotják. Mi az ALGOL nyelvet a hivatkozási nyelv szintjén tárgyaljuk.

Az ALGOL 60 hivatkozási nyelvből hiányoznak azok az utasítások, amelyek a számológépek perifériális egységeire (kiíró, beolvasó, mágnesszalagos stb. egységek) tartalmaznának hivatkozást. E téren az egységesítés komoly akadálya a jelenleg forgalomban levő számológéptípusok nagymértékű különbözősége. A perifériális berendezésekre hivatkozó egységes utasítások beiktatása az ALGOL nyelvbe mind ez ideig nincs véglegesen megoldva, bár már jelentek meg az irodalomban konkrét javaslatok e kérdés megoldására is ([17], [18]).

## 3. Alapfogalmak

Ebben a fejezetben — anélkül, hogy pontos matematikai definiciókra törekednénk — röviden ismertetjük azokat a fogalmakat, amelyeket a későbbi tárgyalás folyamán használunk.

Valamely programozási nyelv *szintaxisán* azoknak a formális szabályoknak összességét értjük, amelyeknek feltétlenül teljesülniük kell ahhoz, hogy egy jelsorozatnak, mint e programozási nyelven (esetünkben az ALGOL-ban) írt programnak értelmet lehessen tulajdonítani. Formális szabály például, hogy egy aritmetikai kifejezésben (formulában) nem következhet egymás után két műveleti jel. Így pl. $a+-b$ szintaktikusan hibás. Egy jelsorozatról azt

mondjuk, hogy valamely *szintaktikus egységbe* tartozik, ha eleget tesz azoknak a formális szabályoknak, amelyekkel az illető szintaktikus egységet definiáljuk. Így például $a+b-c$ eleget tesz az aritmetikai kifejezések (formulák) szintaktikus egységére érvényes formális szabályoknak, ezért beletartozik ebbe a szintaktikus egységbe.

Egy programozási nyelv *szemantikai* szabályainak összességén azoknak az értelmezési szabályoknak összességét értjük, amelyek meghatározzák az egyes — szintaktikusan eleve helyesnek feltételezett — jelsorozatok értelmét. A szemantikai szabályok azt fejezik ki, hogy egy adott program hatására mit hajt végre az elektronikus számológép, „mit jelent" az adott program. Egy számológépprogram lehet szintaktikusan helyes, és ugyanakkor szemantikusan hibás, aminthogy mondhat valaki egy nyelvtanilag teljesen hibátlan mondatot, amely teljesen mást fejez ki, mint amit az illető közölni akart.

Minden számológépprogram bizonyos meghatározott jelkészletből (ábécéből) vett jelek véges sorozata. Azokat a jeleket, amelyeket valamely programozási nyelven (pl. ALGOL-ban) írt programok összeállítása során használni szabad, az illető programozási nyelv *alapjeleinek* nevezzük. Az alapjelek tartalmazhatják pl. a latin ábécé betűit, számjegyeket, matematikai jeleket, írásjeleket stb. Az ALGOL 60 alapjeleit a könyv végén táblázatban foglaltuk össze.

Gyakran használjuk a továbbiakban a *változó* fogalmát, amit a számológép operatív memóriájának egy rekeszével azonosíthatunk. Az operatív memória rekeszei egy-egy számadat időleges tárolására szolgálnak. Minden olyan esetben, amikor ebbe a memóriarekeszbe valamely számadat kerül beírásra, a memóriarekesz korábbi tartalma törlődik, megsemmisül. Ennek megfelelően, a számológépprogramozásban használt változók felvehetnek bizonyos értékeket — a rekesz pillanatnyi tartalmát (mint a későbbiekben majd látjuk, ez az érték nem okvetlenül számérték) — és a program végrehajtása során ez az érték többször megváltozhat. Minden egyes értékváltozáskor a változó új értéke lép a régi helyébe, és a korábbi érték törlődik. Értékváltozás bekövetkezhet a számológép perifériális egy-

ségeivel való adatcsere következtében (pl. lyukszalagról való beolvasás), vagy pedig oly módon, hogy egy megfelelő programutasítás segítségével arra utasítjuk a gépet, hogy a számítás valamely közbülső eredményét adja át egy meghatározott változónak. Az e célra szolgáló utasításokat *értékadó utasításnak* nevezzük. Például, az

$$x := x + y + 2 \qquad (1.1)$$

matematikai egyenlőségre emlékeztető formula az ALGOL-ban azt jelenti, hogy az $x$ és $y$ változók *pillanatnyi* — vagy ahogy az ALGOL-lal kapcsolatban gyakrabban mondani szokták, *aktuális* — értékével ki kell számítani az $x+y+2$ kifejezés számszerű értékét, és ezt az értéket át kell adni az $x$ változónak. Ha $x$ pillanatnyi értéke* 1.2 és $y$ pillanatnyi értéke 2.5, akkor az adott kifejezés értéke 5.7. A felírt értékadó utasítás hatására a számológép kiszámítja ezt a számértéket, és az eredményt átadja az $x$ változónak, aminek következtében $x$ korábbi értéke törlődik, és a következőkben $x$ aktuális értéke már 5.7. Az (1.1) formulában szereplő := jel az ALGOL egyik alapjele, és az értékadást jelöli. Az értékadó utasítások a program legfontosabb alkatrészei, mert segítségükkel tudjuk a közbülső és a végső eredményeket kiszámítani.

A számológépprogram egyes utasításait a gép általában felírásuk sorrendjében hajtja végre. Ha ettől a természetes sorrendtől el kívánunk térni, külön utasításokat kell felírnunk, amelyek kijelölik a soron következő utasítást. Az ilyen utasításokat *vezérlésátadó utasításoknak* nevezzük. A vezérlésátadás (vagyis a soron következő utasítás kijelölése) lehet feltétel nélküli, vagy pedig valamely logikai döntés eredményétől függő. Logikai feltétel lehet pl. két mennyiség egymáshoz való viszonyának vizsgálata: ha mondjuk az $x$ mennyiség kisebb, mint $y$, akkor a természetes sorrend marad érvényben, ha pedig nagyobb vagy egyenlő vele, akkor következőként át kell térni egy meghatározott utasításra. A vezérlésátadó utasításokban

* Tekintettel arra, hogy az ALGOL nyelv nem tizedesvesszőt, hanem tizedespontot használ, az egyszerűség kedvéért a szövegben mindenütt ezt követjük.

valami módon meg kell nevezni azt az utasítást, amellyel a program végrehajtását folytatni kívánjuk. Ebből a célból a megfelelő utasítást ún. *címkével* látjuk el, és a vezérlésátadó utasításban erre a címkére hivatkozunk. Az ALGOL-ban a címkét a hozzá tartozó utasítástól kettőspont választja el. Az (1.1) utasítás, címkével ellátva, az ALGOL 60 szintaktikus szabályainak megfelelően, a következő alakot öltheti:

$$A: x: = x + y + 2 \tag{1.2}$$

Itt a nagy *A* betű az utasítás címkéje, és ez a címke lehetővé teszi, hogy a program más helyén egy vezérlésátadó utasítás hivatkozzék erre az utasításra. A

**go to** *A* (1.3)

ALGOL utasítás hatására például a vezérlés feltétel nélkül átadódik az (1.2) utasításra, azaz következőként a gép ezt az utasítást hajtja végre. Az értékadó és vezérlésátadó utasításokat részletesen a harmadik fejezetben tárgyaljuk.

Gyakran találkozunk a későbbiekben az *azonosító* fogalmával. Az azonosítók a program különböző objektumainak elnevezésére szolgálnak. Így például az *x* betűvel a program egy meghatározott változóját jelöljük, és ennek segítségével különböztetjük meg ezt a változót a program összes többi változójától. Hasonló szerepe van az (1.2) utasításban az *A* betűnek, amely az (1.2) utasítás megnevezésére szolgál. Azonosítóval látjuk el a programban nemcsak a változókat és a címkéket, hanem a program egyéb objektumait is, pl. vektorokat, mátrixokat, a program egyes önmagában zárt egységeit, az ún. eljárásokat stb.

A fordítóprogramnak ahhoz, hogy az ALGOL-ban írt programot le tudja fordítani a számológép elemi utasításainak sorozatára (belső kódjára), szüksége van bizonyos felvilágosításokra a programban szereplő azonosítók jellegére, felhasználásuk módjára, a programban betöltött szerepére vonatkozólag. Ilyen felvilágosítás pl. az, hogy az azonosító a program milyen objektumát (egyszerű változó, vektor, mátrix, címke stb.) jelöli meg; vektorváltozók esetén a vektor elemeinek száma; a változók által felvehető értékek típusa

(pl. csak egész) stb. A program elején, vagy meghatározott egyéb helyein — de mindenesetre az illető azonosító felhasználása előtt — az egyes azonosítókat különböző szempontok szerint csoportosítva felsorolják, és az ily módon keletkező listákból a fordítóprogram útmutatást kap az illető azonosító kezelésére vonatkozólag. Az ilyen listákat az azonosítók *deklarációinak* nevezzük. A deklarációk megszerkesztésével a negyedik fejezetben foglalkozunk.

Befejezésül, előzetes példaként, bemutatunk egy egészen rövid ALGOL programot, egy kétváltozós függvény tabellázásának programját. Tegyük fel, hogy a számológépen megoldandó feladat az

$$f(x, y) = \frac{1}{\sqrt[3]{1 + 2x^2 + 3y^3}} \tag{1.4}$$

kétváltozós függvény tabellázása az $1.2 \leqq x \leqq 12$, $3.6 \leqq y \leqq 36$ négyszögben, mindkét változóban tizedenként. A függvényértékek minden szóba jövő $(x, y)$ értékpárra kinyomtatandók, a hozzájuk tartozó $x$ és $y$ értékekkel együtt.

A fenti feladat ALGOL 60 nyelven írott programja a következő lehet:

```
begin real x, y, f;                                              (1.5)
    x:=1.2;   E: if x>12 then go to B;
    y:=3.6;   D: if y>36 then go to C;
    f:=1/(1+2×x↑2+3×y↑3)↑0.333333;
    print (x, y, f);   y:=y+0.1;   go to D;
C:x:=x+0.1;   go to E;

B: stop end
```

Az (1.5) program első sora a deklarációs rész, a programban szereplő azonosítók listaszerű felsorolása. A **real*** szó arra utal, hogy a felsorolt változók tetszőleges valós számot felvehetnek értékként. Ezután következik a program első értékadó utasítása, amely beállítja az $x$ változó kezdő-

* A könyv végén megtalálható az összes félkövéren szedett angol szó magyar jelentése.

értékét, a tabellázandó intervallum bal oldali végét, 1.2-t. Ezt egy logikai vizsgálat követi arra vonatkozóan, hogy vége-e a számításnak. Ha nem, beállítódik az $y$ változó kezdőértéke is, és újabb logikai vizsgálat következik, amely azt kérdezi, hogy rögzített $x$ mellett kiszámításra került-e már a függvényérték minden $y$-ra? Ha nem, akkor kiszámítódik a függvényérték a pillanatnyi $x$ és $y$ értékekkel, kinyomtatódik (*print* utasítás), megnövelődik $y$ értéke 0.1-gyel, és megismétlődik az $y$-ra vonatkozó logikai vizsgálat (*D* címkéjű utasítás). Ha rögzített $x$ mellett a gép elvégezte a számítást minden szóba jövő $y$-ra, akkor az $x$ változó értéke megnő 0.1-gyel, és megismétlődik az $x$-re vonatkozó logikai vizsgálat (*E* címkéjű utasítás). Ha már minden előírt $x$-re is befejeződött a számítás, a számológép megáll a program végén elhelyezett *stop* utasításon.

Az aritmetikai kifejezések írásában, mint látjuk, a szokványos matematikai írásmódhoz képest eltérések tapasztalhatók. Így a hatványkitevő kisbetűs írása és félsoros emelése helyett, a hatványozást az ALGOL-ban külön alapjel, a felfelé mutató nyíl (↑) jelöli. A harmadik gyök vonását 0.3333 ... valós kitevőre emeléssel helyettesítjük, ahol az $\frac{1}{3}$ törtet olyan pontossággal kell tizedestörtként venni, ahogy azt a számítás pontossága megkívánja. Az aritmetikai kifejezések írásmódjával részletesen a második fejezetben foglalkozunk.

Még megjegyezzük, hogy a *print* és a *stop* utasítások nem részei az ALGOL 60 hivatkozási nyelvnek, de a gépi reprezentációs nyelvek ilyen és ezekhez hasonló utasításokkal ki szokták egészíteni a hivatkozási nyelvet. Ezek a kiegészítések, csakúgy, mint a hivatkozási nyelvhez képest előírt megszorítások, az egyes konkrét gépek fordítóprogramjaihoz mellékelt kézikönyvekben találhatók meg. A ki- és bevitel, valamint a perifériális egységekkel való adatcsere kérdéseire egészen röviden az ötödik fejezet 7. pontjában térünk ki.

*Második fejezet*

# ARITMETIKAI ÉS LOGIKAI KIFEJEZÉSEK

## 1. Bevezetés

Az ALGOL-utasítások leggyakrabban használt elemei, építőkövei az aritmetikai és logikai kifejezések. Ezek önmagukban még nem utasítások, de mivel bármely ALGOL-utasítás tartalmazhat kifejezéseket, megszerkesztési és értelmezési szabályaikkal az utasítások tárgyalása előtt külön foglalkozunk.

Az *aritmetikai kifejezés* ALGOL-fogalma a matematikai képlet (formula) közismert fogalmának ALGOL-nyelvi megfelelője. Az aritmetikai kifejezés számokból és változókból áll, és a számokon, valamint a változók aktuális értékein végzett aritmetikai műveletek eredményeként, egyetlen számértéket állít elő. Ezt a számértéket az aritmetikai kifejezés *aktuális értékének* nevezzük. A kifejezés aktuális értéke a benne szereplő változók értékeinek függvénye. Ezért mindig feltesszük, hogy a kifejezés egy program kiragadott darabja, és hogy ez a program értéket adott a kifejezést felépítő változóknak, mielőtt annak kiszámítására sor kerülne. Ha egy aritmetikai vagy logikai kifejezésben van olyan változó, amelynek a program a kifejezést megelőzőleg nem adott értéket, akkor ez a kifejezés nem számítható ki az adott pillanatban, és aktuális értéke nincs definiálva.

Az említett feltételezés magában foglalja azt is, hogy kiszámítás után a kapott aktuális értékkel még történik valami. Az ALGOL-ban egy kifejezés értéke ugyanis valami módon mindig felhasználásra kerül (pl. ha a kifejezés értékadó utasítás jobb oldalán áll, akkor ezzel az értékkel egyenlővé tétetik a bal oldali változó). A kifejezések tárgyalása során sem azt nem vizsgáljuk, hogy mi módon kapták a

benne szereplő változók az értéküket, sem azt, hogy a kiszámítás után mi lesz az aktuális érték sorsa.

A *logikai kifejezés* fogalma kevésbé közismert, mint az aritmetikai kifejezésé. A logikai kifejezések valamely igennel vagy nemmel megválaszolható kérdésre adnak választ. Logikai feltételeknek főként a programok feltételes elágazatásainál van szerepük. Később látni fogjuk, hogy az aritmetikai és a logikai kifejezések között minden szempontból szoros analógia van, és mindaz, amit aritmetikai kifejezésekről elmondunk, a logikaiakra is érvényes, természetesen helyenként értelemszerű változtatásokkal. Logikai kifejezés aktuális értéke az a válasz („igaz" vagy „hamis"), amit a kifejezés a feltett kérdésre ad. A logikai kifejezések tárgyalásakor ugyanazon feltevésekkel élünk, mint az aritmetikaiaknál.

## 2. Számok

Aritmetikai kifejezések, képletek felírásakor igen gyakran van szükség számkonstansok alkalmazására. Formulákban a betűváltozók mellett csaknem mindig előfordulnak konkrét számok is. Ezért az ALGOL nyelvnek lehetőséget kell biztositania arra, hogy a programokban számkonstansok szerepelhessenek.

A számkonstans nem változtatja értékét, és nem írható értékadó utasítás bal oldalára.

Az ALGOL 60 szempontjából kétféle típusú számértéket különböztetünk meg: „egész" és „valós" számértékeket.* Az egész típusú mennyiségek — mint a nevük is mutatja — csak egész számokat vehetnek fel értékként. Megköveteljük tőlük, hogy a köztük végzett aritmetikai műveletek pontosak legyenek, tehát a gép belsejében ábrázolt eredménynek nem szabad különböznie az eredmény matematikai értékétől. A valós típusú mennyiségek — amelyek értékként tetszőleges, a gépben ábrázolható számot felvehetnek — mindig kerekített értéknek tekintendők, és valós típusú mennyiségek között végzett aritmetikai művelet

* Egyes programozási nyelvek ennél többféle mennyiséggel is dolgoznak: komplex, dupla pontosságú stb.

eredménye a művelet eredményének matematikai értékétől eltérést mutathat. A valós és egész típusú mennyiségek gépen belüli ábrázolása eltérő, és a valós típusú mennyiségek ábrázolási határai rendszerint jóval tágabbak, mint az egész típusú mennyiségeké. A hivatkozási nyelvben nincs megkötés a számábrázolás határaira, és ott a valós mennyiségeket is végtelen pontosnak tekintjük. A valós és egész típusú mennyiségek megkülönböztetése a gépi reprezentáció szempontjai miatt szükséges.

A számkonstansok is kétfélék: egészek és valósak. Az egész számkonstansok tetszőleges számú decimális számjegyből és előjelből állnak. Pozitív egészek előjele tetszés szerint kiírható vagy elhagyható. Azokat az egész számokat, amelyek elé a pozitív előjelet nem írtuk ki, előjeltelen egésznek nevezzük. Az egész számok elé tetszés szerinti számban írhatunk „értéktelen" zérusokat.

Példa egész típusú konstansokra:

$$1, \quad +1, \quad -133, \quad 0008, \quad 92, \quad +1965.$$

Példáink között három közönséges és három előjeltelen egészet mutattunk be.

A valós típusú konstansok négy részből: előjelből, egész részből, törtrészből és decimális normáló tényezőből állnak. Tipikus példa valós konstansra a következő:

$$\underbrace{-}_{\text{előjel}} \quad \underbrace{133}_{\substack{\text{egész}\\\text{rész}}} \quad \underbrace{.0028}_{\text{törtrész}} \quad \underbrace{{}_{10}-22}_{\substack{\text{decimális}\\\text{normáló}\\\text{tényező}}} \tag{2.1}$$

A (2.1) konstans a lehető legáltalánosabb típusú. A számkonstansokból a négy alkotóelem közül bármelyik hiányozhat, de csak teljes egészében. Külön felhívjuk a figyelmet arra, hogy a tizedespont a törtrészhez tartozik, és vagy vele együtt van jelen, vagy vele együtt hiányzik. Hasonlóan a ${}_{10}$ alapjel a decimális normáló tényező szerves része. A pozitív előjel mind a szám elején, mind a kitevőrészben tetszés szerint kiírható vagy elhagyható.

Példák valós típusú konstansokra:

$$-.2789 \qquad (2.2.1)$$

(hiányzik az egész rész és a decimális normáló tényező);

$$92.5042 \qquad (2.2.2)$$

(hiányzik az előjel és a decimális normáló tényező);

$$-55_{10}-12 \qquad (2.2.3)$$

(hiányzik a törtrész).

Példák szintaktikusan hibás konstansokra:

$$+192. \quad 4.5_{10} \quad +._{10}-2$$

(a hibák megkeresését az Olvasóra bízzuk).

A számkonstansok felírására semmiféle normalizálási kényszer nincs. Ugyanazt a számkonstanst a decimális normáló tényező megváltoztatásával és a tizedespont egyidejű áthelyezésével akárhányféleképpen fel lehet írni.

Ebben a pontban 14 ALGOL-alapjelet ismertünk meg. Ezek a következők: a számjegyek 0-tól 9-ig; a + és − jelek; a tizedespont; a hatványalap jele: $_{10}$.

Az ALGOL 60 nem tesz megkötést a legnagyobb megengedett számkonstansra nézve. A gépi reprezentánsok azonban igen. Emiatt igen nagy vagy igen kicsiny abszolút értékű számkonstansokat ALGOL-programokba csak bizonyos óvatossággal szabad beírni, továbbá másoktól átvett ALGOL-programokat meg kell vizsgálni abból a szempontból, hogy nem szerepel-e bennük meg nem engedett konstans. Az ALGOL fordítóprogramok eltérő módon reagálnak a számábrázolás határain kívül eső konstansokra; van olyan, amelyik megáll és hibát jelez, van olyan, amelyik automatikusan a megengedett legnagyobb számmal helyettesíti őket.

## 3. Betűk, változók és azonosítók

Az ALGOL 60 nyelvben alapjelként használhatók az angol ábécé kis- és nagybetűi. Ez összesen $2 \times 26 = 52$ újabb alapjelet jelent. A betűket az ALGOL nyelvben főként azonosítók képzésére használjuk.

Az ALGOL-jelentés megemlíti, hogy az angol ábécé alkalmazása nem kötelező a felhasználók részére; az egyes gépi reprezentánsokban az ábécé tetszés szerint szűkíthető vagy bővíthető. Végső soron az alkalmazott ábécét a gép beviteli jelkészlete (a gépbe bevihető szimbólumok összessége) szabja meg, és ez lényegében műszaki kérdés.

Bár a hivatkozási nyelven írt ALGOL-programokban a szerzők általában tartani szokták magukat az 52 jelből álló ábécé használatához, mi a továbbiakban — kényelmi okokból — a magyar (ékezetes betűkkel bővített) ábécét fogjuk használni. Felhívjuk azonban az Olvasó figyelmét, hogy a publikálásra szánt — különösen az idegen nyelven publikálandó — programokban a magyar ékezetes betűk alkalmazását ajánlatos kerülni.

Az azonosító fogalma az I. fejezet 3. pontjából ismert. Azonosítóval vannak ellátva a változók, továbbá a program azon utasításai, amelyekre a program egyéb helyén hivatkozás történik. Később meg fogunk ismerni még néhány szintaktikus egységet, amelynek azonosítója van. (Ilyenek a tömbök, IV. fejezet 3. pont, a kapcsolók, IV. fejezet 4. pont és az eljárások, V. fejezet.)

Miután az ALGOL 60 nyelv 52 betűje nem biztosít elegendő különböző azonosítót, meg van engedve több jelből álló azonosítók képzése is. A pontos szintaktikus szabály a következő: *azonosító lehet bármely betűkből és számjegyekből álló, betűvel kezdődő jelsorozat*. A hivatkozási nyelvben tetszés szerinti helyen tetszés szerinti számú szóköz vagy sorváltás helyezhető el, így lehetőség van több szóból álló azonosítók képzésére is. Példák azonosítókra:

*A3, X5Y, xrst, Kiss János.*

A szóközök és a sorváltások száma közömbös, vagyis pl. *Kiss János* egyenértékű a következőkkel:

*KissJános, Kiss János, Ki ssJános, Kiss*
*János,*

stb. Az *ALGOL nyelv nem ismer elválasztási szabályt. A program bármely helyén, vagyis bármely jel után át lehet*

*térni új sorra, vagy bárhol kihagyható tetszőleges számú szóköz.* Egyes gépi reprezentánsok azonban nem engedik meg az azonosítók széttagolását.

Itt megjegyezzük, hogy vannak az ALGOL-ban olyan *alapjelek*, amelyek látszólag több jelből állnak. Ilyen például a $_{10}$, a:=, valamint az ún. szóalapjelek: **true, if, do** stb. Ezek *a szintaxis szempontjából egyetlen jelnek számítanak, és semmilyen módon nem választhatók szét.*

Az azonosítók hosszára a hivatkozási nyelv nem tesz megkötést. Megengedettek tetszés szerinti számú jelből álló azonosítók. Egyes gépi reprezentánsokban az azonosítók hossza korlátozva van meghatározott számú jelben (általában 5...8 jelet szoktak megengedni).

Az ALGOL nyelv szempontjából valós, egész és logikai típusú változókat különböztetünk meg. Az egész típusú változók csak egész számokat vehetnek fel értékként. A valós típusúak bármely valós számot. A logikai vagy más néven Boole-féle változók értékként nem számokat, hanem az „igaz" és a „hamis" ítéletek valamelyikét veszik fel. Ez utóbbiakat részletesen a 6. pontban tárgyaljuk.

Azt, hogy egy meghatározott azonosítóval megjelölt változó egész, valós vagy logikai típusú-e, a program deklarációs részéből tudjuk meg. ALGOL-programban csakis deklarált változók fordulhatnak elő. A deklarációk megadásának szabályaival, és a deklarációkra vonatkozó egyéb kérdésekkel a IV. fejezetben foglalkozunk. A könyvnek ebben és a következő fejezetében minden előforduló változóról feltesszük, hogy deklarálva van.

Az ALGOL-ban nemcsak magukban álló skaláris változókat használunk, hanem dolgozhatunk összetartozó változócsoportoknak: vektoroknak, mátrixoknak stb. elemeivel is. Ezeknek az egy csoportba tartozó változóknak közös azonosítójuk és egy vagy több megkülönböztető indexük van. Az indexet az azonosító után, szögletes zárójelbe írjuk. Pl.: *a*[3,2]. Most az egyszerűség kedvéért csakis index nélküli skaláris változókkal dolgozunk. Az indexes változók használatát a következő (III.), deklarálásukat pedig a IV. fejezetben tárgyaljuk.

## 4. Aritmetikai műveletek

Az aritmetikai kifejezések tárgyalásának további előkészítéseként ebben a pontban az ALGOL 60 nyelvben definiált aritmetikai műveleteket ismertetjük.

Mielőtt a különféle aritmetikai műveletek részletes tárgyalására térnénk, előrebocsátjuk, hogy egy aritmetikai művelet eredménye az ALGOL-ban akkor van egyértelműen definiálva, ha számértékével együtt a típusát (valós vagy egész) is megadjuk. A továbbiakban több szabályt fogunk megismerni arra nézve, hogy milyen esetekben milyen típusú eredmény keletkezik.

Az ALGOL nyelvben hatféle aritmetikai művelet van definiálva. Ezek a következők:

1. Összeadás. Jele: +
2. Kivonás. Jele: −
3. Szorzás. Jele: ×
4. Osztás. Jele: /
5. Egészek osztása. Jele: ÷ (olv.: „mínusz-kétpont")
6. Hatványozás. Jele: ↑ (olv.: „nyíl")

A felsorolt valamennyi műveleti jel ALGOL-alapjel.

Az összeadás, a kivonás és a szorzás a megszokott módon vannak definiálva. Eredményük akkor és csak akkor egész típusú, ha mindkét műveleti komponens egész típusú; minden más esetben valós.

A / jellel jelölt osztási művelet mindig valós eredményt állít elő, még akkor is, ha mindkét komponens egész típusú. Ezt a fontos szabályt a gyakorlatlan programozók sokszor elfelejtik, és ez néha rendkívül nehezen felderíthető programhibákra vezet.

A ÷-tal jelölt osztási művelet csakis egész típusú komponensekre van értelmezve, és mindig egész típusú eredményt állít elő. A művelet számszerű eredményét az

$$a \div b = sign\,\frac{a}{b} \times entier\left|\frac{a}{b}\right| \tag{2.3}$$

képlet segítségével definiáljuk, ahol sign $x$ az előjelfüggvény:

$$sign\ x = \begin{cases} -1, \text{ ha } x<0 \\ 0, \text{ ha } x=0 \\ +1, \text{ ha } x>0 \end{cases}$$

*entier* $x$ pedig az egészrész-függvény: *entier* $x$ azt a legnagyobb egész számot jelenti, amely $x$-nél még nem nagyobb. Pl. $entier(-3)=-3$, $entier(+3.1)=+3$, $entier(-3.1)=-4$.

A (2.3) formula szavakban annyit jelent, hogy az egészek osztása

1. követi az osztás jól ismert előjelszabályát,
2. az eredményben törlődik a tizedespont utáni rész.

Példák:

$$13 \div (-4) = -3, \qquad 7 \div 2 = 3.$$

A 0-val való osztás egyik fajta osztási műveletre sincs értelmezve, ilyen esetben általában a célprogram* futtatása közben a gép megáll.

Az aritmetikai műveletek közül a hatványozás a legbonyolultabb. Az 1. ábra táblázatosan összefoglalja a hatványozás műveleteinek különböző eseteit. Az ott részletezett szabályok közül a legfontosabbak a következők:

1. Egész kitevős hatványozást ismételt szorzásokkal (negatív kitevő esetén ismételt szorzásokkal és reciprokképzéssel) kell végezni.

2. Ha a hatványkitevő valós típusú, akkor a hatványozás az

$$a^x = e^{x \times \ln a} \tag{2.4}$$

azonosság alapján végzendő. Mivel a logaritmusfüggvény (a valós függvények körében) csak pozitív argumentumra van értelmezve, zérus vagy negatív alapnak valós típusú kitevőre való hatványozása értelmetlen.

3. A hatványozás művelete akkor és csakis akkor ad egész típusú eredményt, ha az alap egész típusú, és a kitevő

* Célprogramon a fordítóprogram működése után kapott, a számológép számára közvetlenül végrehajtható gépi programot értjük.

nemnegatív egész. Jól jegyezzük meg: ha egy hatványozási műveletben a kitevő *változó*, tehát *nem szám*, akkor az eredmény típusát befolyásolhatja a kitevőnek nemcsak

KITEVŐ ($x$)

| ALAP ($a$) | | Egész | | | Valós | | |
|---|---|---|---|---|---|---|---|
| | | $>0$ | $=0$ | $<0$ | $>0$ | $=0$ | $<0$ |
| Valós | $>0$ | $a \times a \times \ldots \times a$<br>valós | 1<br>valós | $\frac{1}{a \times a \times \ldots \times a}$<br>valós | $e^{x \ln a}$<br>valós | 1<br>valós | $e^{x \ln a}$<br>valós |
| | $=0$ | 0<br>valós | NINCS ÉRT. | NINCS ÉRT. | 0<br>valós | NINCS ÉRT. | NINCS ÉRT. |
| | $<0$ | $a \times a \times \ldots \times a$<br>valós | 1<br>valós | $\frac{1}{a \times a \times \ldots \times a}$<br>valós | NINCS ÉRT. | NINCS ÉRT. | NINCS ÉRT. |
| Egész | $>0$ | $a \times a \times \ldots \times a$<br>egész | 1<br>egész | $\frac{1}{a \times a \times \ldots \times a}$<br>valós | $e^{x \ln a}$<br>valós | 1<br>valós | $e^{x \ln a}$<br>valós |
| | $=0$ | 0<br>egész | NINCS ÉRT. | NINCS ÉRT. | 0<br>valós | NINCS ÉRT. | NINCS ÉRT. |
| | $<0$ | $a \times a \times \ldots \times a$<br>egész | 1<br>egész | $\frac{1}{a \times a \times \ldots \times a}$<br>valós | NINCS ÉRT. | NINCS ÉRT. | NINCS ÉRT. |

1. ábra

típusa, hanem aktuális értéke is! (Az összes többi aritmetikai műveletnél az eredmény típusát megszabja a komponensek típusa, az aktuális értéktől függetlenül.)

## 5. Feltétlen* aritmetikai kifejezések

Ebben a pontban az aritmetikai kifejezésekkel kapcsolatos szintaktikai és szemantikai szabályokat tárgyaljuk. Először az aritmetikai kifejezések egyes szűkebb osztályait ismertetjük, majd a tárgyalás folyamán fokozatosan bővítjük a vizsgált fogalomkört.

Először olyan aritmetikai kifejezéseket tekintünk, amelyek nem tartalmaznak sem zárójelet, sem függvényoperációt (sin, cos, entier stb.), sem indexes változót.

A fenti kikötésnek eleget tevő aritmetikai kifejezések szintaktikus szempontból operandusok és aritmetikai műveleti jelek váltakozó sorozatai. *Operanduson számkonstanst vagy változó-azonosítót értünk.* Az aritmetikai kifejezés vagy egy operandussal kezdődik, vagy a + és − jelek valamelyikével.

Példák:

$$alfa + beta/gamma\uparrow 3 \tag{2.5.1}$$

$$-x/y + Zr - Zt + 1 \tag{2.5.2}$$

$$i \div j + k \div n \tag{2.5.3}$$

A szintaktikus definíciót megadtuk. Áttérünk most a fenti típusú jelsorozatok értelmezési szabályaira. Ebben az elemi algebrából jól ismert szabályokat kell — a szokásosnál kissé precízebben megfogalmazva — átültetnünk az ALGOL 60 terminológiájára. Mint ahogy itt is, és a későbbiekben is látni fogjuk, az ALGOL nyelvét úgy szerkesztették meg, hogy minden szabály lehetőleg kézenfekvő legyen.

A hatféle műveleti jelet (lásd a 4. pontot) három csoportba soroljuk. Az egyes csoportokon belül a műveleti jelek egyenrangúak. Különböző csoportba tartozó műveleti jelek között elsőbbségi (idegen szóval: precedencia-) szabályok vannak értelmezve.

* Az aritmetikai kifejezések legáltalánosabb típusát az ún. feltételes aritmetikai kifejezések alkotják, ezek tárgyalását l. a harmadik fejezet 4. p.-ban.

Az egyes műveletcsoportok a következők:

1. Elsődleges műveleti jel a hatványozás jele: ↑
2. Másodlagosak az ún. multiplikatív műveleti jelek: ×, /, ÷.
3. Harmadlagosak az ún. additív műveleti jelek: +, −.

A precedenciaszabály értelmében egy aritmetikai kifejezésben minden operandus a vele szomszédos operandusok közül ahhoz tartozik, amelyhez a magasabb precedenciájú műveleti jel kapcsolja. Ez a szabály megfelel az ismert elemi szabálynak; pl. az

$$a+b\times c \tag{2.6}$$

kifejezésben a $b$ operandus a vele szomszédos $a$ és $c$ operandusok közül $c$-hez tartozik, mert a szorzásjel mint multiplikatív műveleti jel, magasabb precedenciájú az additív + jelnél. Az aritmetikai kifejezések elején és végén álló operandusokat úgy kell tekinteni, mintha „külső" oldalukon elsőbbségi szempontból még az additívnál is gyengébb műveleti jel állna.

Az egyenlő precedenciájú műveleti jelek esetéről egy második szemantikai szabály rendelkezik, az ún. „balról jobbra" szabály. Ennek értelmében egy operandus, melynek két oldalán egyenértékű műveleti jelek állnak, a tőle balra levő operandushoz tartozik.

A precedenciaszabály és a balról jobbra szabály együttesen bármely (ideiglenes megszorításainknak eleget tevő) aritmetikai kifejezés kiszámítására tartalmaz utasítást. A teljes kiszámítási szabály röviden úgy szól, hogy balról jobbfelé tartó sorrendben végig kell vizsgálni az aritmetikai kifejezés műveleti jeleit, és vagy a balról jobbra szabály, vagy a precedenciaszabály szerint kell a műveleteket végezni, aszerint, hogy a szomszédos műveleti jelek azonos vagy különböző precedenciájúak-e.

*Példa.* Tekintsük a következő aritmetikai kifejezést:

$$a+b+c\times d\uparrow 2. \tag{2.7}$$

Legyen az összes változó valós típusú, és legyenek az aktuális értékek rendre: $a=2$, $b=3$, $c=4$ és $d=2$. (2.7) kiszámí-

tása a következőképpen megy végbe: vizsgáljuk rendre a kifejezés műveleti jeleit. *a*, mint bal szélső változó, automatikusan *b*-hez tartozik. *b* két oldalán azonos precedenciájú + jelek állnak. Itt a balról jobbra szabály érvényesül: összeadjuk *a*-t és *b*-t. Az eredmény 5. A *c* változó bal oldalán +, jobb oldalán × jel áll; az utóbbi magasabb rendű, és így *c*-t a jobb oldali szomszédjához kell kapcsolni. Azonban a jobb oldali szomszéd, *d*, jobb oldalán még magasabb rendű műveleti jel áll, a hatványozás jele, így *d* is jobb oldali szomszédjához tartozik a precedenciaszabály értelmében. Ez a jobb oldali szomszéd a 2-es, ennek jobb oldalán nincs műveleti jel. Ily módon a $d \uparrow 2$ művelet elvégezhető, az eredmény 4. Ezt a számot összeszorozhatjuk *c*-vel, az eredmény 16, és ezt hozzáadva $a+b = 5$-höz, végeredményként 21-et kapunk. Az összes közbülső művelet eredménye, és a végeredmény is valós típusú.

Az elsőbbségi és a balról jobbra szabályokról mondottakat másképpen is megfogalmazhatjuk.

Mint mondottuk, az aritmetikai kifejezés operandusok és aritmetikai műveleti jelek váltakozó sorozata. Egy adott aritmetikai kifejezést felosztható több részkifejezésre oly módon, hogy különválasztjuk az eredeti kifejezésnek azon részeit, amelyek nem tartalmaznak additív (harmadlagos) műveleti jelet. Egy ilyen, additív műveleti jelet nem tartalmazó részkifejezést *tagnak* nevezünk. Az aritmetikai kifejezés tagokból és az ezeket összekapcsoló additív műveleti jelekből áll. Természetesen előfordulhat, hogy egy aritmetikai kifejezés egyáltalán nem tartalmaz additív műveleti jelet, ebben az esetben egyetlen tagból áll. Például a (2.7) kifejezés három tagból áll, ezek rendre *a*, *b*, $c \times d \uparrow 2$.

Az egyes tagokon belül ismét elhatárolhatunk olyan részeket, amelyek nem tartalmaznak multiplikatív (másodlagos) műveleti jelet, ezeket *tényezőnek* nevezzük. Minden tag tényezőkből és az őket összekapcsoló multiplikatív műveleti jelekből áll. Ha egy tag egyáltalában nem tartalmaz multiplikatív műveleti jelet, akkor egyetlen tényezőből áll. Így pl. a (2.7) kifejezés tagjai közül csak a harmadik áll egynél több tényezőből: ezek *c* és $d \uparrow 2$.

Az egyes tényezők még mindig tartalmazhatnak elsődleges műveleti jelet. Két szomszédos elsődleges műveleti jel (↑) között azonban már nem állhat semmiféle műveleti jel, csak operandus. Az operandusokat másképpen *elsődleges kifejezésnek* nevezzük.

Az aritmetikai kifejezések kiszámítása úgy történik, hogy balról jobbra tartó sorrendben kiszámítjuk az egyes tagokat, és az újonnan kiszámított tagot hozzáadjuk az előző (tőle balra levő) tagok részletösszegéhez. Az egyes tagokon belül is balról jobbra tartó sorrendben halad a számítás: egyenként kiszámítjuk az egyes tényezőket, és ezeket rendre „hozzászorozzuk" az előző (balra levő) tényezők részletszorzatához. Minden egyes tényező kiszámításán belül, balról jobbra haladó sorrendben végzett hatványozásokon keresztül jutunk el a tényező aktuális értékéhez.

A balról jobbra szabály egyik következménye, hogy az $a\uparrow b\uparrow c$ aritmetikai kifejezés szokásos matematikai jelölésben $a^{b^c}$-nek felel meg. (Ugyanis $a\uparrow b$ eredményét kell a $c$-edik hatványra emelni.)

Másik következménye a balról jobbra szabálynak, hogy az $a/b\times c$ ALGOL-kifejezés közönséges matematikai jelölésben $\frac{a}{b}\times c$-nek, nem pedig $\frac{a}{b\times c}$-nek felel meg. Az utóbbi zárójelek felhasználása nélkül $a/b/c$ alakban írható át ALGOL-kifejezéssé.

*Példák*

**1.** Írjuk át ALGOL-kifejezéssé a következő képletet:

$$i+j+\frac{a\times b}{k^j}. \tag{2.8}$$

*Megoldás:*

$$i+j+a\times b/k\uparrow j \tag{2.9}$$

**2.** Melyek a (2.9) kifejezésben a tagok, a tényezők és az elsődleges kifejezések?

*Válasz:* tagok: $i$, $j$, $a\times b/k\uparrow j$
tényezők: $i$, $j$, $a$, $b$, $k\uparrow j$
elsődleges kifejezések: valamennyi betűváltozó.

*Megjegyzés:* a kifejezés első két tagja egyetlen elsődleges kifejezésből (betűváltozóból) áll, ezek egyszersmind tényezők is. Hasonlóan a kifejezés harmadik tagjában a három tényezőből kettő szintén elsődleges kifejezés (változó), és csak a harmadik áll két elsődleges kifejezésből és egy ↑ műveletből.

**3.** Legyenek a (2.9)-ben szereplő változók aktuális értékei:

$i=2$ típusa egész
$j=3$ típusa egész
$a=3$ típusa valós
$b=8$ típusa valós
$k=2$ típusa egész

Kiszámítandó a kifejezés aktuális értéke és az eredmény típusa.

*Megoldás:*

1. tag aktuális értéke 2 típusa egész
2. tag aktuális értéke 3 típusa egész
Első részletösszeg = 5 típusa egész
3. tag 1. tényező akt. értéke 3 típusa valós
2. tényező akt. értéke 8 típusa valós
Első részletszorzat = 24 típusa valós
2. tényező akt. értéke 8 típusa valós
Második részletszorzat = 3 típusa valós
A 3. tag aktuális értéke 3 típusa valós
Második részletösszeg = 8 típusa valós

A második részletösszeg egyben végösszeg is, az eredmény 8, valós ábrázolásban.

**4.** Ha a (2.9) kifejezésben, az előbbi példa feltevésével ellentétben, valamennyi változó egész típusú lenne, mi lenne az eredmény típusa?

*Válasz:* sem az eredmény számértéke, sem típusa nem változna meg. Ugyanis az $a \times b/k \uparrow j$ tag mindenképpen valós típusú, a benne szereplő osztási művelet miatt, még akkor is, ha a műveletek eredménye véletlenül egész számot állít elő.

**5.** Legyen $i$ egész típusú változó. Írjunk olyan aritmetikai kifejezést, amelynek aktuális értéke $i$, ha $i$ páros, és $i-1$, ha $i$ páratlan.

*Megoldás.* A fentihez hasonló feladatok megoldására gyakran előnyösen alkalmazható az egész osztás ($\div$) művelete. A példa követelményeinek eleget tevő aritmetikai kifejezés:

$$i \div 2 \times 2 \tag{2.10}$$

Verifikáljuk pl. (2.10) viselkedését $i=10$-re, és $i=11$-re! $10 \div 2 = 5$, $2 \times 5 = 10$. Páros $i$-re tehát rendben van a dolog. Páratlan $i$ esetén: $11 \div 2 = 5$, $2 \times 5 = 10$. (2.10) tehát valóban megfelel a feltételeknek.

**6.** Szintaktikusan helyes-e az alábbi jelsorozat az ALGOL-ban:

$$2 \times 10\uparrow 3 \tag{2.11}$$

mi a különbség a fenti kifejezés és a $2_{10}3$ szám között?

*Válasz:* (2.11) szintaktikusan helyes ALGOL-jelsorozat mint aritmetikai kifejezés. Szemantikailag (2.11) annyiban különbözik a $2_{10}3$ számkonstanstól, hogy aritmetikai műveleti jelek szerepelnek benne, és a fordítóprogram (2.11)-et mint kifejezést és nem mint számkonstanst tekinti. Ennek az a következménye, hogy valahányszor a gép a (2.11) konstanshoz ér a számítás folyamán, annak értékét mindannyiszor kiszámítja a benne szereplő 2, 10 és 3 számkonstansokból. Ebből különösebb baj nem származik, de a gépet fölöslegesen dolgoztatjuk, és ezért számoknak (2.11)-hez hasonló felírását ajánlatosabb kerülni.

Az aritmetikai kifejezések eddig vizsgált osztályát most egy kissé kibővítjük, és bevezetjük az ún. *standard függvények* fogalmát. Standard függvényen az ALGOL-ban azokat a leggyakrabban előforduló függvényeket értjük, amelyeket az ALGOL egyezményes azonosítóval lát el, és amelyeknek minden ALGOL-fordítóprogramban rendelkezésre kell állniuk, anélkül, hogy külön definiálni kellene őket.

Standard függvények azonosítói „zárolt" jellegűek, és semmilyen más célra nem használhatók. A standard függvények:

*sin* (*x*), *cos* (*x*), *ln* (*x*), *exp* (*x*), *sqrt* (*x*), *arctan* (*x*), *abs* (*x*), *entier* (*x*), *sign* (*x*).

A *sin* (*x*) és *cos* (*x*) függvények jelentése elemi matematikai jelentésükkel azonos. Az *ln* (*x*) függvény a természetes (*e*-alapú) logaritmust jelenti. Az *exp* (*x*) függvény az *e*-alapú exponenciális függvény, a *sqrt* a négyzetgyökvonás. Az *arctan* függvény az arkusztangens főértékét számítja ki. Az *abs* standard függvény az abszolút érték képzésére szolgál, az *entier* az egészrészt állítja elő. A *sign* az előjelfüggvény, amelynek definícióját az előző pontban, a (2.3) képlettel kapcsolatban adtuk meg.

A fenti standard függvények — az entier kivételével — mind egész, mind valós típusú argumentumra értelmezve vannak. Hét standard függvény valós típusú eredményt állít elő, kettő pedig (a *sign* és az *entier*) egész típusút. Az *entier* standard függvény csak valós típusú argumentumra van értelmezve.

A standard függvények argumentumát az ALGOL-ban minden esetben zárójelbe kell tenni, még akkor is, ha egyetlen operandusból áll. Az argumentum lehet tetszőleges aritmetikai kifejezés.

A standard függvények argumentumukkal együtt az aritmetikai kifejezésekben teljesen úgy viselkednek, mint az operandusok, tehát a változók vagy a számkonstansok. Ezt úgy is kifejezhetjük, hogy egy standard függvény az argumentumával együtt elsődleges kifejezést alkot.

*Példák*

**1.** Írjuk át ALGOL-kifejezéssé az alábbi aritmetikai formulát:

$$\sin x + \cos 2kx + e^{\arctan x}. \tag{2.12}$$

*Megoldás:*

*sin* (*x*) + *cos* (2 × *k* × *x*) + *exp* (*arctan* (*x*))

*Megjegyzés:* a standard függvények akárhányszoros mélységben egymásba skatulyázhatók. Példánkban az arctan függvényt skatulyáztuk bele az exp függvény argumentumába.

**2.** Írjuk át ALGOL-kifejezéssé az alábbi aritmetikai formulát:

$$\sqrt{1-\sin^3 x+e^{-ax}}. \tag{2.13}$$

*Megoldás:*

$$sqrt\,(1-sin(x)\uparrow 3+exp(-a\times x))$$

*Megjegyzés:* mivel a standard függvények teljesen analóg módon viselkednek az operandusokkal, a hatványozás műveletét is ugyanúgy jelöljük, akár változó, akár számkonstans, akár standard függvénykifejezés az alap. A szokványos $\sin^2 x$, $\sin^3 x$ stb. jelöléseknek az ALGOL-ban nincs megfelelőjük.

Most áttérünk a zárójeles kifejezések tárgyalására.

Az elemi matematikából ismeretes, hogy a zárójel a műveletek „természetes" precedenciájának megváltoztatására használatos. A (2.6) kifejezésről például mondottuk, hogy az a $(b\times c)$ szorzatnak $a$-val való összeadását írja elő. Ha ettől eltérő műveleti sorrendet kívánunk előírni, akkor zárójelet kell alkalmaznunk:

$$(a+b)\times c.$$

A zárójel „ellene mond" a $\times$ jel magasabb precedenciájának, és a természetes elsőbbségi szabálytól eltérő műveleti sorrendet ír elő. Az elemi matematikában meg van engedve fölösleges, redundáns zárójelek alkalmazása: (2.6) helyett írhatjuk azt is, hogy $a+(b\times c)$.

Az elemi matematikának ezeket a zárójelezési szabályait szinte minden változtatás nélkül visszük át az ALGOL-ba. A zárójelezés célja itt is a természetes precedencia-sorrendtől eltérő műveleti sorrend előírása, és megengedett fölösleges, redundáns zárójelek alkalmazása is, ALGOL-ban azonban aritmetikai kifejezések zárójelezésére csakis kerek zárójelek alkalmazhatók. A kerek zárójeleket: „("

és „)", a standard függvényekkel kapcsolatban vezettük be új alapjelként. A szögletes zárójelet később majd szintén bevezetjük mint ALGOL-alapjelet, azonban nem aritmetikai zárójelként fogjuk használni. A „kapcsos" zárójel nem szerepel az ALGOL-ban.

Az aritmetikai kifejezések zárójelezéséről az alábbi egyszerű szabály rendelkezik: bármely aritmetikai kifejezés, ha kerek zárójelbe zárják, elsődleges kifejezéssé változik, és közönséges operandussal (számkonstanssal vagy változóval) válik egyenértékűvé. Ha tehát bármilyen bonyolult aritmetikai műveletsorozat eredményével további aritmetikai műveleteket kívánunk végezni, nem kell mást tennünk, mint a megfelelő kifejezést zárójelbe zárnunk, és mint egyetlen operandust kezelnünk. Az aritmetikai zárójelek tetszőleges mélységben egymásba skatulyázhatók. Mintahogy a zárójelbe zárt kifejezés egyetlen operandust képvisel, ahhoz, hogy egy ilyen kifejezéssel további aritmetikai műveletet végezhessünk, előbb ennek aktuális értékét kell ismernünk. A zárójeles kifejezések bevezetésével tehát az előzőkben megismert — az elsőbbségi és a balról jobbra szabályon alapuló — kiszámítási algoritmus csak annyiban bővül, hogy ha egy aritmetikai művelet valamelyik operandusa zárójeles kifejezés, akkor előbb ki kell számítani a zárójelen belüli kifejezés aktuális értékét, és azután kell elvégezni a kijelölt műveletet. Mivel a zárójelek akárhányszoros mélységben egymásba skatulyázhatók, ezt a szabályt általában rekurzíve kell értelmeznünk.

*Példák*

**1.** Írjuk át ALGOL-kifejezéssé az alábbi matematikai formulát:

$$\frac{\sin\left[b(2x^2+1)+1\right]+\cos\left[a^2+b^2(c-d)\right]}{a+\sqrt{(a^2+b^2)x-p}}. \qquad (2.14)$$

*Megoldás:*

```
(sin (b × (2 × x↑2 + 1) + 1) + cos (a↑2 + b↑2 × (c − d)))/(a +
sqrt((a↑2 + b↑2) × x − p))
```

*Megjegyzés:* mint már említettük, az ALGOL-ban nincs elválasztási szabály. Tehát bárhol áttérhetünk újabb sorra, csak egy alapjelen belül nem. (Az áttekinthetőség érdekében célszerű azonban a függvényeket argumentumukkal és a változókat indexükkel lehetőleg egy sorba írni.)

Hosszabb képleteknek ALGOL-kifejezéssé való átírásánál nagyon kell ügyelni arra, hogy minden megkezdett zárójel be legyen zárva. Az ALGOL-lal való munka során leggyakrabban felmerülő szintaktikus hiba a zárójelek téves elhelyezése. Szerencsére ezt a hibát a fordítóprogram még a fordítás stádiumában jelzi.

**2.** Számítsuk ki az alábbi ALGOL-kifejezés aktuális értékét:

$$a\uparrow(b\times c)+d/(x+2\times(a-c)) \qquad (2.15)$$

Legyen valamennyi változó valós típusú, és aktuális értékeik legyenek rendre:

$$a=2;\ b=2;\ c=3;\ d=-50;\ x=14.5.$$

*Megoldás:*

Először is elhatároljuk (2.15) különálló részkifejezéseit. A kifejezés két tagból áll, az első tag $a\uparrow(b\times c)$, a második $d/(x+2\times(a-c))$. Az első tag egyetlen tényező, amely tartalmaz egy hatványozási műveletet. A hatványozás exponensében zárójeles kifejezés áll, mint elsődleges kifejezés. A második tagot két tényező alkotja, az egyik, *d*, egyszerű változó, a második zárójeles (elsődleges) kifejezés. Ez utóbbi tényező ismét két tagú összeg; első tagja egyszerű változó, a második két tényezőből álló szorzat, melynek második tényezője ismét két tagból álló zárójeles kifejezés, tagjai egyszerű változók.

A kifejezés kiszámítása hasonló módon történik, mint a (2.9) kifejezés kiszámítása.

Első tag: az alap 2, a kitevő $2\times 3=6$. A hatvány $2^6=64$. Az első tag aktuális értéke 64, az eredmény valós.

Második tag: számláló $-50$.
nevező első tagja 14.5; második tagja $-2$. A nevező aktuális értéke 12.5; a második tag aktuális értéke $-4$. Az eredmény valós.

A teljes kifejezés aktuális értéke $64-4 = 60$.

E helyen fel kívánjuk hívni a figyelmet a következőre: az ALGOL-fordítóprogram tevékenysége az aritmetikai kifejezések fordításával kapcsolatban arra korlátozódik, hogy az ALGOL-programot lefordítja az adott konkrét gép nyelvére, azaz olyan gépi utasításokat generál, amelyek *végrehajtása* esetén kiszámítódik az aritmetikai kifejezés aktuális értéke. Tehát a fordítóprogram *nem számítja ki ezt az értéket*, és ezért csakis arra képes, hogy megállapítsa egy kifejezésről, *formálisan* helyes-e, de annak szemantikai hibáit csak a célprogram működése közben lehet észrevenni.

A (2.9) és (2.15) ALGOL-kifejezések kiszámítása közben nem választottuk szét élesen, hogy abból, amit csinálunk, mi tartozik a fordítóprogram hatáskörébe, és mi a célprograméba. A kifejezés elemzése, szétbontása tagokra, tényezőkre, elsődleges kifejezésekre, az esetleges számkonstansoknak elhelyezése a memóriában, és általában mindaz, ami nem igényli a változók aktuális értékének ismeretét, a fordítóprogram dolga. Így többek között a balról jobbra szabály és az elsőbbségi szabály is a fordítóprogram megszerkesztésére vonatkozó utasítás. Ezzel szemben az aritmetikai műveletek numerikus elvégzése a célprogram működése közben történik.

Befejezésül felhívjuk a figyelmet néhány igen gyakori szintaktikus hibára, amelyeket a programozók aritmetikai kifejezések felírásakor szoktak elkövetni.

1. Zárójelekkel kapcsolatban: nem egyenlő számú kezdő- és végzárójel. Szögletes vagy kapcsos zárójel alkalmazása. Standard függvény argumentumánál a kötelező zárójel elmulasztása.

2. Szorzásjellel kapcsolatban: szorzásjel elhagyása vagy ponttal jelölése.

3. Egyéb műveleti jelekkel kapcsolatban: két aritmetikai műveleti jel kerül egymás mellé. Ez különösen két esetben szokott előfordulni. Az egyik a hatványozás műveletének helytelen felírása: a hatványozás jele után a kitevő előjeleként még egy műveleti jelet írunk, pl. $a\uparrow -3$. Ilyen esetben feltétlenül zárójelet kell alkalmazni: $a\uparrow(-3)$. A másik gyakori hiba új sorra való áttérés után, az előző sor utolsó műveleti jelének megismétlése a következő sorban. Ez a matematikai képletírástól eltérően, az ALGOL-ban nincs megengedve.

## 6. Logikai változók és logikai műveletek

Az eddigiekben megismert aritmetikai változók aktuális értéke (egész vagy valós) számérték. Ebben a pontban bevezetünk egy újabb változótípust, amely aktuális értékként nem számokat, hanem ún. logikai értékeket vesz fel; ezt *logikai* vagy *Boole-féle változónak* nevezzük. Ehhez szükségünk van két újabb ALGOL-alapjelre, ezek a **true** és a **false** (igaz, ill. hamis).

A **true** és **false** alapjelek formájukban eltérnek az eddig megismert ALGOL-alapjelektől. Ezek különlegesen (félkövéren) szedett angol szavak, amelyek jelentése kifejezi ALGOL nyelvbeli funkciójukat. Az ALGOL-ban mintegy húsz ilyen szóalapjel van definiálva. Ezeket minden szempontból ugyanúgy kell tekinteni, mint az egyetlen szimbólumból álló alapjeleket; az írógép egy-egy külön billentyűjének fogjuk fel őket. Ebből következik, hogy elválaszthatatlanok, és nem azonosak a betűkből összerakott megfelelő szóval (például a „true" nem azonos a „**true**"-val); utóbbiak az azonosítók szintaktikus egységébe tartoznak, és a hivatkozási nyelv meg is engedi alapszavakkal egyező azonosítók képzését, hiszen félreértésre nincs lehetőség, mert az alapszavak és az azonosítók szedése különböző.

Gépelt, sokszorosított szövegekben, kéziratban, továbbá az elektronikus számológépek lyukasztóberendezésein általában nincs kétféle betűtípus (pl. normál és félkövér). Ennek hiányában az ALGOL-alapjeleket jelentő

szavakat aláhúzással kell megkülönböztetni. Sajnos van olyan ALGOL-reprezentáns is, amelyben egyáltalán nincs lehetőség az alapjelek és az azonosítók megkülönböztetésére, még aláhúzással sem, és ez maga után vonja egy olyan helyi megkötés bevezetését, hogy nem szabad szóalapjelekkel megegyező azonosítókat használni.

Visszatérve a **true** és **false** alapjelekre, azokat *logikai értékeknek* nevezzük. A logikai értékek nem számok; számváltozó nem vehet fel értékként logikai értéket és fordítva, logikai változó nem vehet fel értékként számértéket. Mint a továbbiakban látni fogjuk, a **true** és **false** értékeknek a logikai (igennel vagy nemmel megválaszolható) feltételek ALGOL-beli leírásában jut nagy szerep.

Az aritmetikai kifejezésekhez hasonlóan, a logikai kifejezések tárgyalása előtt a logikai értékek között értelmezett műveleteket kell ismertetnünk.

Elöljáróban megjegyezzük, hogy a logikai értékek között értelmezett műveleteket e könyv keretében, és általában az ALGOL nyelv keretében, önmagukban definiáljuk, és nem foglalkozunk ezeknek a műveleteknek a matematikai logikával való kapcsolatával. Ugyanez vonatkozik a később tárgyalandó logikai kifejezésekre is.

Az ALGOL 60 nyelvben ötféle logikai művelet van értelmezve. Ezek a következők:

1. Logikai szorzás (konjunkció, olvasva: „és"), jele: $\wedge$;
2. Logikai összeadás (diszjunkció, olvasva: „akár"), jele: $\vee$;
3. Implikáció (olvasva: „implikálja"), jele: $\supset$;
4. Ekvivalencia (olvasva: „ekvivalens"), jele: $\equiv$;
5. Negáció (olvasva: „nem"), jele: $\neg$.

Valamennyi logikai műveleti jel ALGOL-alapjel.

A logikai értékek közötti műveleteket ún. igazságérték-táblázatok segítségével értelmezzük; ezek a művelet paramétereinek minden lehetséges kombinációjára megadják a művelet eredményét.

1. Logikai szorzás: $a \wedge b$

| | | $b$ | |
|---|---|---|---|
| | | **true** | **false** |
| $a$ | **true** | **true** | **false** |
| | **false** | **false** | **false** |

(2.16)

A (2.16) táblázat szerint az $a \wedge b$ művelet eredménye minden olyan esetben **false,** amikor a paraméterek közül legalább az egyik **false**.

2. Logikai összeadás: $a \vee b$

| | | $b$ | |
|---|---|---|---|
| | | **true** | **false** |
| $a$ | **true** | **true** | **true** |
| | **false** | **true** | **false** |

(2.17)

A (2.17) táblázat szerint az $a \vee b$ művelet eredménye minden olyan esetben **true**, amikor a paraméterek közül legalább az egyik értéke **true**.

3. Implikáció: $a \supset b$

| | | $b$ | |
|---|---|---|---|
| | | **true** | **false** |
| $a$ | **true** | **true** | **false** |
| | **false** | **true** | **true** |

(2.18)

A (2.18) táblázat szerint az $a \supset b$ művelet eredménye csakis akkor **false**, ha $a$ értéke **true**, de $b$ értéke **false**. Minden más esetben a művelet eredménye **true**.

4. Ekvivalencia: $a \equiv b$

| | | $b$ | |
|---|---|---|---|
| | | **true** | **false** |
| $a$ | **true** | **true** | **false** |
| | **false** | **false** | **true** |

(2.19)

A (2.19) táblázat szerint az $a \equiv b$ művelet eredménye akkor **true**, ha $a$ és $b$ egyező logikai értékűek, és akkor **false**, ha $a$ és $b$ értéke különböző.

5. Negáció: $\neg a$

A negáció — az előző négy logikai művelettel ellentétben — egyparaméteres művelet, és a logikai érték tagadását, ellenkezőre fordítását jelenti.

| | $a$ | |
|---|---|---|
| | **true** | **false** |
| $\neg a$ | **false** | **true** |

(2.20)

## 7. Relációk

Ebben a pontban a logikai kifejezések egy további fontos alkotórészét, a *reláció* fogalmát ismerjük meg.

Szintaktikus szempontból a reláció két feltétlen aritmetikai kifejezés (lásd 5. p.), relációjellel összekapcsolva. A relációjelek a következők:

| | | | |
|---|---|---|---|
| $=$ | (egyenlő) | $<$ | (kisebb) |
| $\geqq$ | (nagyobb vagy egyenlő) | $\leqq$ | (kisebb vagy egyenlő) |
| $>$ | (nagyobb) | $\neq$ | (nem egyenlő) |

Valamennyi relációjel ALGOL-alapjel.

**Példák relációra:**

$$x \geqq 3_{10} - 3 \tag{2.21.1}$$

$$a + b/(b - c) < 2 \times (k + 2) \tag{2.21.2}$$

$$sqrt\,(3 + x \uparrow 2) > x/y \tag{2.21.3}$$

Magában álló változó vagy számkonstans aritmetikai kifejezésnek számít, így (2.21.1) szintaktikus szempontból korrekt reláció. Sőt, éppen ez az egyszerű típusú reláció kerül az ALGOL-ban leggyakrabban felhasználásra.

Szemantikai szempontból a relációnak mindig egy logikai értéket tulajdonítunk, melyet a változók aktuális értéke határoz meg. A reláció aktuális értéke **true**, ha a jobb és bal oldali aritmetikai kifejezés aktuális értéke között a relációjel által kijelölt egyenlőtlenség (= jel esetén egyenlőség) valóban fennáll, és **false**, ha a relációjel által kijelölt egyenlőtlenség (egyenlőség) nem áll fenn.

Például, ha az $x$ változó aktuális értéke 0.0035, akkor a (2.21.1) reláció aktuális értéke **true**, ha $x$ aktuális értéke 0.0025, akkor (2.21.1) aktuális értéke **false**.

Nincs kikötve, hogy a reláció jobb és bal oldalán álló kifejezések aktuális értéke azonos típusú legyen. Ha például $k$ egész típusú változó, akkor (2.21.2) jobb oldala egész típusú, bal oldala valós.* A jobb és bal oldali kifejezéseknek azonban kiszámíthatóknak kell lenniük, azaz bennük csakis deklarált, és értékkel rendelkező változók szerepelhetnek.

Ha $a$, $b$ és $c$ valós típusú változók, és aktuális értékeik rendre 6.2; 2.1; 1.5, továbbá $k$ aktuális értéke 2, akkor (2.21.2) bal oldalának aktuális értéke 9.7, a jobb oldal aktuális értéke 8, a reláció aktuális értéke **false**.

Viszont $x = 4$; $y = 2$ esetén (2.21.3) aktuális értéke **true**.

Az egyenlőség jelével (=) meghatározott relációval kapcsolatban szeretnénk felhívni a figyelmet egy gyakori hibaforrásra. Amikor egy egyenlőség valamelyik oldalán valós értékű kifejezés áll, ez a gépi számábrázolás véges

* A bal oldali kifejezés típusa, függetlenül az $a$, $b$, $c$ változók típusától, mindig valós, a kifejezésben szereplő / műveleti jel miatt (l. a 4. pontot).

volta miatt igen kicsiny, de zérustól különböző eltérést mutathat a kifejezés pontos értékétől. Ennek következtében előfordulhat, hogy két aritmetikai kifejezést, amelyeknek elvben egyenlőnek kellene lenniük egymással, a gép nem egyenlőnek érzékel, és a reláció **true** értéke helyett látszólag tévesen, **false** választ ad. Miután előre nem tudhatjuk, hogy egy adott aritmetikai kifejezést a konkrét számítás során a gép milyen irányban fog „elkerekíteni", gépre szánt ALGOL-programokban nem célszerű valós típusú mennyiségek közötti egyenlőséget alkalmazni. Az egyenlőséget könnyen átírhatjuk egyenlőtlenséggé oly módon, hogy a bal és jobb oldal aktuális értékének különbségére valamilyen megszorítást teszünk, például azt, hogy az kisebb legyen valamilyen előre megadott mennyiségnél.

Például az

$$a = b + x/(1 - y) \tag{2.22}$$

egyenlőséget tanácsosabb

$$abs\,(a - b - x/(1 - y)) \leqq {}_{10} - 5 \tag{2.23}$$

alakban felírni. A különbségre tett megszorítás mértéke természetesen az adott programtól függ.

Egész típusú mennyiségek veszélytelenül összehasonlíthatók az egyenlőség jelének segítségével.

## 8. Feltétlen* logikai kifejezések

A 6. és 7. előkészítő pontok után most áttérünk a feltétlen logikai kifejezések tárgyalására.

Amint arra korábban már utaltunk, az aritmetikai és a logikai kifejezések között messzemenő analógia áll fenn, és minden, amit az aritmetikai kifejezésekről elmondottunk, értelemszerű változtatásokkal átvihető a logikaiakra. A továbbiakban ez az analógia lesz a tárgyalás vezérfonala.

A tárgyalást itt is a zárójelet nem tartalmazó kifejezésekkel kezdjük. Szintaktikus szempontból egy zárójelet

* A logikai kifejezések legáltalánosabb típusát az ún. feltételes logikai kifejezések alkotják, ezek tárgyalását lásd a III. fejezet 4. p.-ban.

nem tartalmazó logikai kifejezés műveleti jelek és operandusok váltakozó sorozata. Műveleti jelen itt a kétparaméteres logikai műveletek műveleti jeleit értjük: $\wedge$, $\vee$, $\supset$, $\equiv$.

Az operandus értelmezésében eltérés van az aritmetikai és logikai kifejezések között. Az aritmetikai esetben operanduson számkonstanst vagy változót értettünk. Mindkettőnek megvan a logikai megfelelője: a számkonstansnak megfelel a logikai érték (**true** és **false**), az aritmetikai változónak a logikai változó. A logikai operandus szintaktikus egységébe ezeken kívül azonban egy harmadik szintaktikus egység is beletartozik, nevezetesen a reláció. A relációnak nincs aritmetikai megfelelője.

Példák feltétlen logikai kifejezésekre:

$$x \geqq 2.5 \wedge x < 3.3 \qquad (2.24.1)$$

$$A \wedge B \equiv U \vee \textbf{true} \qquad (2.24.2)$$

$$x \uparrow 2 + y \uparrow 2 + 1 \leqq p + 1 \vee p = 2 \vee B \qquad (2.24.3)$$

A (2.24.1) kifejezés két operandusból áll, mindkettő reláció. A (2.24.2) kifejezés első három operandusa logikai változó,* a negyedik logikai érték. (2.24.3) operandusai közül az első kettő reláció, a harmadik logikai változó.

A logikai kifejezések szemantikáját ugyanazok a törvények szabályozzák, mint az aritmetikaiakét, nevezetesen a precedenciaszabály és a balról jobbra szabály. A logikai műveleti jelek precedenciasorrendje a következő:

elsődleges a logikai szorzás: $\wedge$
másodlagos a logikai összeadás: $\vee$
harmadlagos az implikáció: $\supset$
negyedleges az ekvivalencia: $\equiv$

(A negáció jele különleges elbánást kapott az ALGOL-ban, ezzel később foglalkozunk.)

Az aritmetikai kifejezések kiszámítása úgy megy végbe, hogy amint azt az 5. p.-ban láttuk, felbontjuk a kifeje-

* A jelen pontban a nagy latin betűket *logikai*, a kis latin betűket *aritmetikai* típusú változók azonosítóiként fogjuk használni.

zést tagokra, tényezőkre és elsődleges kifejezésekre, és ezek kiszámításánál a balról jobbra szabály szerint járunk el. A logikai kifejezések esetében a kiszámítási szabály hasonló. Az itt szereplő szintaktikus egységek neveit nem érdemes megjegyezni, jelöljük azokat egyszerűen *A*, *B*, *C* és *D*-vel. *A* típusú részkifejezésnek nevezzük a logikai kifejezés azon részeit, amelyek nem tartalmaznak negyedleges műveleti jelet (≡). A logikai kifejezés *A* típusú részkifejezésekből és az ezeket összekapcsoló ≡ jelekből áll. Ha a logikai kifejezés nem tartalmazza az ekvivalencia jelét, akkor egyetlen *A* típusú részkifejezésből áll.

Az *A* típusú részkifejezéseket felbontjuk olyan *B* típusú részkifejezésekre, amelyek nem tartalmazzák az implikáció jelét (⊃). Az *A* típusú részkifejezések *B* típusú részkifejezésekből és az őket összekapcsoló ⊃ jelekből állnak. Ha valamelyik *A* típusú részkifejezés nem tartalmazza az implikáció jelét, akkor egyetlen *B* típusú részkifejezésből áll.

A *B* típusú részkifejezéseket tovább bontjuk olyan *C* típusú részkifejezésekre, amelyek nem tartalmazzák a logikai összeadás műveleti jelét (∨). A *B* típusú részkifejezések *C* típusúakból és az őket összekapcsoló ∨ jelekből állnak.

Végül a *C* típusú részkifejezéseket felbontjuk olyan *D* típusú részkifejezésekre, amelyek semmiféle logikai műveleti jelet nem tartalmaznak. *D* típusú részkifejezések a logikai változók, a relációk, és a logikai értékek — tehát éppen az operandusok. A *D* típusú részkifejezés pontos megfelelője az aritmetikai elsődleges kifejezésnek.

*Példa*

Tekintsük a következő logikai kifejezést:

$$A \vee B \wedge x - y \geqq p \times q \vee x > 2 \supset y > 2 \equiv B \wedge C \vee D \wedge E \equiv \textbf{false} \tag{2.25}$$

A (2.25) kifejezésnek a fentiek szerinti felbontását lásd a 2. ábrán!

A logikai kifejezés kiszámítása úgy történik, hogy balról jobbra tartó sorrendben rendre kiszámítjuk az *A*, ezeken belül a *B*, ismét ezeken belül a *C*, és ismét ezeken belül

| *A* típusú részkifejezések | *B* típusú részkifejezések | *C* típusú részkifejezések | *D* típusú részkifejezések |
|---|---|---|---|
| $A \vee B \wedge x-y \geqq p \times q \vee x>2 \supset y>2$ | $A \vee B \wedge x-y \geqq p \times q \vee x>2$ | $A$ | $A$ |
| | | $B \wedge x-y \geqq p \times q$ | $B$ |
| | | | $x-y \geqq p \times q$ |
| | | $x>2$ | $x>2$ |
| | $y>2$ | $y>2$ | $y>2$ |
| $B \wedge C \vee D \wedge E$ | $B \wedge C \vee D \wedge E$ | $B \wedge C$ | $B$ |
| | | | $C$ |
| | | $D \wedge E$ | $D$ |
| | | | $E$ |
| **false** | **false** | **false** | **false** |

2. ábra

a *D* típusú részkifejezéseket, és az eredményt komponáljuk az előző azonos típusú részkifejezések kiszámítása során nyert részeredménnyel. Komponáláson itt az ekvivalencia, az implikáció, a logikai összeadás vagy a logikai szorzás műveletének elvégzését értjük, aszerint, hogy *A*, *B*, *C* vagy *D* típusú részkifejezések kiszámításáról van szó.

Számítsuk ki pl. a (2.25) kifejezés aktuális értékét! Legyenek a változók aktuális értékei a következők:

Logikai változók:

$A =$ **false**

$B =$ **false**

$C =$ **false**

$D =$ **false**

$E =$ **false**

Aritmetikai változók:

$x = 2.5$

$y = 1.5$

$p = 1.1$

$q = 2.2$

A számítás menete:

$A =$ **false**;

$B =$ **false**;

$x - y \geqq p \times q =$ **false**;

**false** $\wedge$ **false** = **false**;

**false** $\vee$ **false** = **false**.

Eljutottunk az első logikai részletösszeghez, ennek eredménye **false**.

$x > 2 =$ **true**;

**false** $\vee$ **true** = **true**.

Hozzáadtuk $x > 2$ aktuális értékét az előbb nyert logikai részletösszeghez, és ezzel végeztünk az első *B* típusú részkifejezés kiszámításával; az eredmény **true**.

$y > 2 =$ **false**.

Elérkeztünk az ekvivalencia jeléhez. A második *B* típusú részkifejezés egyetlen relációból állt, ennek értéke **false**. A két *B* típusú részkifejezés aktuális értéke között elvégez-

zük az implikáció műveletét; a (2.18) igazságtáblázat szerint az eredmény **false**, és ez egyben az első *A* típusú részkifejezés aktuális értéke is.

A $B \wedge C$ és $D \wedge E$ logikai szorzatok mindketten **false** értékűek, ezek logikai összege ismét **false**. Megint egy $\equiv$ jelhez érkeztünk, és eközben nem találtunk $\supset$ jelet, így a második *A* típusú részkifejezés egyetlen *B* típusúból áll, és aktuális értéke **false**.

Alkalmazzuk most az ekvivalencia műveletét az első és második *A* típusú részkifejezésre. A (2.19) igazságtáblázat szerint **false** $\equiv$ **false** = **true**, így az első két *A* típusú részkifejezés kiszámítása után egy **true** részeredményt kaptunk.

A harmadik *A* típusú részkifejezés egyetlen logikai értékből (**false**) áll, így azonnal alkalmazhatjuk a második $\equiv$ műveletet is; a (2.19) táblázat szerint ennek aktuális értéke **false**, és ezzel a kifejezés aktuális értékének kiszámítása befejeződött.

Foglalkozzunk most a negáció műveletével. Mint láttuk, a negáció egyparaméteres művelet, amely az operandus értékét ellenkezőjére fordítja. Zárójel nélküli logikai kifejezésekben a $\neg$ jel vagy a kifejezés elején, mint legelső szimbólum foglal helyet, vagy valamelyik kétparaméteres művelet jele után áll. Szemantikai szempontból egy operandus, az előtte álló negációjellel együtt, *D* típusú részkifejezést alkot. Ezt a szabályt másképpen úgy is megfogalmazhatjuk, hogy a negáció jele mindig az utána álló operandust tagadja. Ez valójában azt jelenti, hogy a negáció jele valamennyi logikai művelet jelénél magasabb precedenciát élvez, azonban mivel a precedencia fogalmát eddig csak kétparaméteres műveletekre értelmeztük, szükségesnek tartottuk a negáció jelének szerepét más oldalról is megvilágítani.

*Példa*

Tekintsük a következő logikai kifejezést:

$$\neg A \vee B \wedge \neg C \equiv \neg -x + y < b - a \qquad (2.26)$$

Legyenek az aktuális értékek a következők: $A$ = **true**, $B$ = **true**, $C$ = **false**, $x = -3.27$, $y = 2.48$, $b = 11.8$, $a = 5.42$.

A kifejezés két *A* típusú részkifejezésre bontható. Az első *A* típusú részkifejezés kiszámításának menete a következő:

$\neg A =$ **false**, az első logikai részletösszeg **false**.

$B =$ **true**, $\neg C =$ **true**, **true** $\wedge$ **true** = **true**.

Alkalmazzuk most a logikai összeadás műveletét az előző logikai részletösszeg és a most kapott részeredmény között: **false** $\vee$ **true** = **true**. Elérkeztünk a $\equiv$ jelig, véget ért az első *A* típusú részkifejezés kiszámítása, az eredmény **true**.

A második (a $\equiv$ jel utáni) *A* típusú részkifejezés egyszerű reláció. A reláció bal oldalán álló aritmetikai kifejezés aktuális értéke 5.75, a jobb oldalán álló kifejezésé 6.38, így a reláció aktuális értéke **true**, ennek „negált" értéke **false**. Alkalmazzuk az ekvivalencia műveletét az első és a második *A* típusú részkifejezés aktuális értéke között: **true** $\equiv$ **false** = **false**. A (2.26) kifejezés aktuális értéke tehát **false**. A negáció jeléről elmondottakkal összhangban, a negáció műveletét a kiszámítás közben az *A*, ill. *C* operandusokra, valamint a logikai kifejezés jobb oldalán álló relációra alkalmaztuk, és a negált értékekkel végeztük a további kétparaméteres műveleteket.

Az aritmetikai kifejezésekkel fennálló analógia alapján könnyen áttérhetünk a zárójeles kifejezésekre is. A zárójel most is elsősorban a precedenciaszabály megtörésére való: olyankor alkalmazzuk, amikor valamely operandust a gyengébben kötő műveleti jelhez akarunk csatolni. Bármely logikai kifejezés zárójelbe zárva *D* típusú kifejezést alkot, azaz úgy viselkedik, mint egyetlen operandus. Mielőtt egy zárójeles kifejezés aktuális értékével további műveletet végeznénk, előzőleg a zárójelen belüli összes műveletet el kell végeznünk. A zárójeles kifejezések bevezetésével az eddig megismert számítási szabályok csak az elemi algebrából ismert zárójelfelbontási szabállyal bővülnek. A zárójelek tetszőleges mélységben egymásba skatulyázhatók, és egymásba skatulyázott zárójelek esetén a zárójelfelbontási szabályt rekurzíve kell alkalmazni. Redundáns zárójelek tetszés szerint alkalmazhatók.

Megjegyezzük még, hogy az ALGOL hivatkozási nyelvben nincs definiálva logikai értéket előállító standard

függvény, így a standard függvények fogalmának nincs logikai megfelelője.

*Példa*

Tekintsük a következő logikai kifejezést:

$$((A \vee \neg B) \wedge (C \vee \neg D) \supset (E \equiv F)) \wedge G \qquad (2.27)$$

Legyenek az aktuális értékek a következők: $A =$ **true**, $B =$ **false**, $C =$ **false**, $D =$ **true**, $E = F =$ **false**, $G =$ **true**. Kiszámítandó (2.27) aktuális értéke.

Először a belső zárójeleket bontjuk fel: $A \vee \neg B =$ **true**. $C \vee \neg D =$ **false**. A két belső zárójeles kifejezés aktuális értéke között logikai szorzást hajtunk végre: **true** $\wedge$ **false** = **false**. A következő belső zárójel felbontása során egy **true** részeredményt kapunk, ui. **false** $\equiv$ **false** = **true**. Alkalmazzuk az implikáció műveletét az előző és a jelenlegi részeredmény között: **false** $\supset$ **true** = **true**. Most már rendelkezésünkre áll a külső zárójelen belüli kifejezés aktuális értéke; ezzel és $G$ aktuális értékével logikai szorzást végzünk: **true** $\wedge$ **true** = **true**. (2.27) aktuális értéke tehát **true**.

*Két megjegyzés:*

1. Egy logikai kifejezés aktuális értékének meghatározásához bizonyos esetekben szükségtelennek látszik végigszámolni a teljes kifejezést. Ha pl. egy logikai szorzási művelet első komponense **false** értékű, akkor a második komponens aktuális értékétől függetlenül a kifejezés aktuális értéke **false** lesz. Hasonló esetek előfordulnak a logikai összeadás és az implikáció műveleteivel kapcsolatban is. Az efféle lehetőségeknek géppel való vizsgáltatása azonban körülményes, nem éri meg a fordító- és a célprogram bonyolítását. Ezenfelül az ALGOL hivatkozási nyelv kategorikusan meg is tiltja a kifejezések kiszámításának ilyenfajta lerövidítését. A tilalom okát a későbbiekben (V. fejezet, 4. p.) fogjuk megismerni.

2. Az $A \supset B$ logikai kifejezéssel egyenértékű a következő: $B \vee \neg A$. Az $A \equiv B$ kifejezést pedig a következővel helyettesíthetjük: $A \wedge B \vee \neg A \wedge \neg B$. Tehát az implikáció és az ekvivalencia műveletei visszavezethetők a logikai

összeadásra és szorzásra. Ilyen átírásoknál azonban mindig ügyelni kell a műveleti jelek elsőbbségére, és ahol szükséges, újabb zárójeleket kell bevezetni.

Az aritmetikai és logikai kifejezések közti analógiát, illetve az ettől való eltéréseket az alábbi összehasonlító táblázattal szemléltethetjük:

| *Aritmetikai kifejezés* | *Logikai megfelelője* |
|---|---|
| Számkonstans | Logikai érték |
| Aritmetikai változó | Logikai változó |
| — | Reláció |
| Hatványozás | — |
| Multiplikatív műveletek | Logikai szorzás |
| Additív műveletek | Logikai összeadás |
| — | Implikáció |
| — | Ekvivalencia |
| — | Negáció |
| Elsődleges kifejezés | *D* típusú részkifejezés |
| Tag, tényező | *A*, *B* és *C* típusú részkifejezések |
| Kezdő és végzárójel | Kezdő és végzárójel |
| Standard függvény | — |
| Balról jobbra szabály | Balról jobbra szabály |
| Elsőbbségi (precedencia-) szabály | Elsőbbségi (precedencia-) szabály |

## 9. Gyakorlatok

**1.** Írjuk át ALGOL-kifejezéssé, a lehető legkevesebb zárójel alkalmazásával, az alábbi matematikai formulákat:

*a)*
$$\frac{a\sin(2nx)+b\cos(2nx)}{ax^2+2bxy+cy^2}$$

*b)*
$$\frac{(a+b)^3+\ln(x^2+y^2)}{\sqrt[3]{x-y}\cdot e^{-\frac{x^2-y^2}{2x^3}}}$$

c)
$$\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+x}}}}$$

d)
$$\frac{\sqrt[n]{\alpha+\beta}\,\sqrt[m]{\gamma-\delta}}{\sqrt[l]{\alpha+\gamma}\,\sqrt[k]{\beta-\delta}}.$$

**2. Írjuk vissza szokványos alakba az alábbi ALGOL-kifejezéseket:**

a) $q\uparrow p\uparrow r\uparrow s/a\uparrow b\uparrow c\uparrow d \times x\uparrow u\uparrow v\uparrow z$

b) $sqrt(x+y+z)\times((a\uparrow 4+b\uparrow 4)\uparrow 0.25+(c\uparrow 4+d\uparrow 4)\uparrow 0.25)$
$/sqrt(1+exp(-gamma\times(a+2\times b))+alfa\uparrow 2\times b)$

c) $arctan((1+ln((x\uparrow 2+y\uparrow 2-z\uparrow 2)/(p\uparrow 2-q\uparrow 2+s\uparrow 2)))/(1+$
$exp((b\uparrow 2+c\uparrow 2-a\uparrow 2)/(g\uparrow 2+e\uparrow 2-f\uparrow 2))))$

**3. Keressük meg a fölösleges zárójeleket az alábbi ALGOL-kifejezésekben:**

a) $(a+b)+(c\times d)\times(e\uparrow f)\uparrow g+(x/y)/p$

b) $((x\uparrow 2)\times(y\uparrow 2))/sqrt((x-2)-(a\uparrow b)\uparrow c)$

c) $a/((b-c)+d\times sin(((2\times a)+(3\times b))/(b-(2\times c))))$

**4. Keressük meg a szintaktikus hibákat az alábbi kifejezésekben:**

a) $exp((2a\uparrow 2-2b\uparrow 2)/(a2\uparrow 2-b2\uparrow 2))$

b) $(sin\,x+cos\,x)/(2\times a\times x+2\times b\times x)$

c) $-x+_2-4\times a\uparrow 2-y/2+b-x;$

d) $(-b+sqrt(b\uparrow 2-4\times a\times c))/2/a$

e) $X\times[a+b\times c(c\uparrow 2+d\uparrow 2)]$

f) $(x-y)/2.(a-b)\uparrow 2$

g) $alfa-beta/gamma+3_{10}i+j$

**5. Számítsuk ki az alábbi aritmetikai kifejezések aktuális értékét, és határozzuk meg az eredmény típusát:**

a) $x\uparrow 2+sqrt(1-a-x)\times b\uparrow c\uparrow d/3$

A változók aktuális értékei legyenek rendre:
$x=3.5;\quad a=-4.46;\quad b=3;\quad c=3.5;\quad d=2;$
és legyen valamennyi változó valós típusú.

b) $arctan(x-2\times y)\times 4\times a/(a-b)+1/(1-t)$

A változók aktuális értékei:
$x=2.48$; $y=0.74$; $a=0.5$; $b=-4.5$; $t=0.6$.
Valamennyi változó valós típusú.

*c)* $2 \times a \div (i+j) + (b-a)/2 \times x$
$a=3$ valós $i=1$ egész
$b=4.28$ valós $j=2$ egész
$x=5.03$ valós

*d)* $x\uparrow 2 + 2 \times a \times x \times y + y\uparrow 2 + sqrt(sin(2 \times x)) + a\uparrow(y-x)$
$a=-3.28$; $x=21.3$; $y=10.7$.
Mindhárom változó valós típusú.

*e)* $sqrt(i+j) \div k$
$i=18$; $j=7$; $k=5$.
Mindhárom változó egész típusú.

**6.** Legyen $i$ egész típusú változó. Írjunk feltétlen ALGOL aritmetikai kifejezéseket, amelyekre

*a)* $f_1(i) = \begin{cases} i, \text{ ha } i \text{ páratlan} \\ i+1, \text{ ha } i \text{ páros} \end{cases}$

*b)* $f_2(i) = \begin{cases} i, \text{ ha } i \text{ páratlan} \\ i-1, \text{ ha } i \text{ páros} \end{cases}$

*c)* $f_3(i) = \begin{cases} i, \text{ ha } i \text{ páros} \\ i+1, \text{ ha } i \text{ páratlan} \end{cases}$

**7.** Legyenek $i$ és $m$ egész típusú változók. Feltesszük, hogy $i \leqq m$. Írjunk feltétlen ALGOL aritmetikai kifejezéseket, amelyek értéke:

*a)* $f_1(i) = \begin{cases} i+1, \text{ ha } i \leqq m-1 \\ 1, \text{ ha } i=m \end{cases}$

*b)* $f_2(i) = \begin{cases} i-1, \text{ ha } i \geqq 2 \\ m, \text{ ha } i=1 \end{cases}$

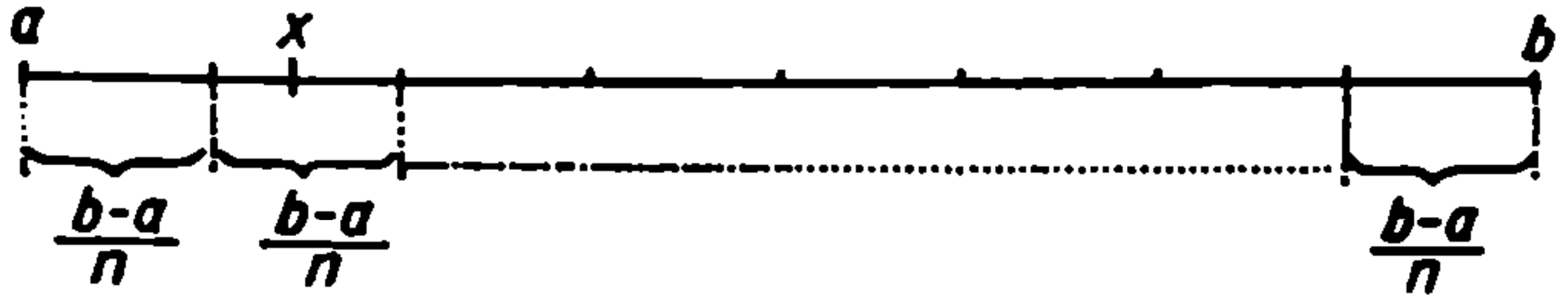
**8.** Legyenek $i, m, j$ egész típusú változók. Feltesszük, hogy $1 \leqq i \leqq m$ és $1 \leqq j \leqq m$. Írjunk feltétlen aritmetikai kifejezéseket, amelyekre

*a)* $f_1(i) = \begin{cases} i+j, \text{ ha } i+j \leqq m \\ i+j-m, \text{ ha } i+j > m \end{cases}$

*b)* $f_2(i) = \begin{cases} i-j, \text{ ha } i-j \geqq 1 \\ i-j+m, \text{ ha } i-j < 1 \end{cases}$

A 7. feladat ennek speciális esete, $j=1$-re.

**9.** Legyenek $a$, $b$ és $x$ valós és $n$ egész típusú változó. Feltesszük, hogy $a \leqq x \leqq b$. Felosztjuk az $[a, b]$ intervallumot $n$ egyenlő részre. Az $x$ változóról nem tudjuk, hogy e részintervallumok közül hányadikban van. A feladat olyan aritmetikai kifejezés felírása, amelynek aktuális értéke éppen annak a részintervallumnak az alsó határa, amelyikben $x$ benne van. (Lásd az ábrát!)




3. ábra

**10.** Számítsuk ki a következő feltétlen logikai kifejezések aktuális értékeit:

*a)* $A \vee B \supset C \wedge D \equiv i - j = 2$

Legyenek a változók aktuális értékei:

$A =$ **false** $\quad i = 3$
$B =$ **true** $\quad j = 2$
$C =$ **true**
$D =$ **false**

*b)* $a \leqq x \wedge x \leqq b \wedge c \leqq y \wedge y \leqq d \equiv x < y$
$a = 2.5; \quad b = 3.5; \quad c = 10.3; \quad d = 10.5; \quad x = 4.2; \quad y = 3.9.$

*c)* $((A \equiv B) \supset (C \equiv D)) \wedge \neg A \wedge \neg D$

$A =$ **true** $\quad C =$ **false**
$B =$ **false** $\quad D =$ **true**

*d)* $A \vee x \uparrow 2 + y \uparrow 2 \geqq 1 \supset x < 1$
$A =$ **false**
$x = 0.5; \quad y = 0.3.$

*e)* $A \wedge \neg B \vee (C \equiv D) \equiv B \vee \neg A \wedge \neg C \supset D$

$A =$ **true** $\quad C =$ **false**
$B =$ **false** $\quad D =$ **true**

**11.** Keressük meg a redundáns zárójeleket az alábbi kifejezésekben:

*a)* $((A \equiv B) \vee (x \uparrow 2 \times p \times q - 3.5_{10}2 > 0)) \supset (\neg A \wedge \neg B) \vee (\neg C \wedge \neg D) \equiv$
$\equiv (C \wedge D)$

*b)* $((A \vee B) \supset ((C \wedge \neg A) \vee \neg B) \supset C) \wedge D$
*c)* $(((A \wedge B) \vee C) \supset D) \equiv E$
*d)* $(((A \equiv B) \supset C) \vee D) \wedge E$

**12.** Írjunk logikai kifejezéseket, amelyeknek aktuális értéke **true**, ha $i$ osztója $j$-nek és **false**, ha nem osztója ($i$ és $j$ egész típusú változók).

*Harmadik fejezet*

# UTASÍTÁSOK

## 1. Bevezetés

Ebben a fejezetben az ALGOL nyelv különböző utasítástípusait fogjuk megismerni. Az egyes utasítástípusokat egy-egy külön pontban tárgyaljuk. Ezekben viszonylag kevesebb a szöveg közti példa, mert az illusztratív példákat külön pontba vontuk össze (8. p.). Ismételten felhívjuk az Olvasó figyelmét a gyakorlatokra (9. p.) is.

Az ALGOL 60 nyelvben a következő utasítástípusok vannak definiálva:

1. Értékadó utasítás (l. 2. p.);
2. Vezérlésátadó utasítás (l. 3. p.);
3. Feltételes utasítás (l. 4. p.);
4. Üres utasítás (l. 4. p.);
5. Összetett utasítás (l. 5. p.);
6. Ciklusutasítás (l. 7. p.);
7. Blokk;
8. Eljárásutasítás.

A blokkokat a IV. fejezetben, az eljárásutasításokat az V. fejezetben több pontban tárgyaljuk. A vezérlésátadó utasítások tárgyalása nem fejeződik be a III. fejezetben, a helymegjelölő kifejezések tárgyalásának kapcsán a IV. fejezetben ismét visszatérünk rájuk.

## 2. Értékadó utasítások

Az I. fejezet 3. p.-ban megismertük az értékadó utasítás fogalmát, amely általában egy bal oldali változóból, és egy aritmetikai kifejezésből áll. Az értékadó utasítás végre-

hajtása közben kiszámításra kerül a jobb oldali kifejezés aktuális értéke, és ezt az értéket felveszi a bal oldali változó, esetleges korábbi értékének egyidejű törlése mellett.

Az ALGOL nyelvben — mint azt az előzőkben láttuk — aritmetikai és logikai típusú változók vannak definiálva. Ennek megfelelően megkülönböztetünk aritmetikai és logikai értékadó utasításokat. Az aritmetikai értékadó utasítások jobb oldali kifejezése aritmetikai kifejezés, és bal oldali változója aritmetikai változó. A logikai értékadó utasításokban a jobb oldali kifejezés logikai kifejezés, és a bal oldali változó logikai változó.

Az értékadás jelölésére az ALGOL-ban külön alapjelet vezettek be, a := (olvasva: kettőspont-egyenlő) jelet.

Legegyszerűbb esetben az ALGOL értékadó utasítás egy bal oldali változóból, az értékadás jeléből és egy jobb oldali (aritmetikai vagy logikai) kifejezésből áll:

$$V := K \tag{3.1}$$

Ha $K$ aritmetikai kifejezés, akkor a bal oldali $V$ változónak aritmetikai típusúnak kell lennie, ha pedig $K$ logikai kifejezés, akkor $V$ logikai változó.

Példák értékadó utasításokra:

$$x := a \times x \uparrow 2 + 2 \times b \times x \times y + c \times y \uparrow 2 \tag{3.2.1}$$

$$A := i = k \wedge p = q \tag{3.2.2}$$

A (3.2.1) és (3.2.2) utasítások szemantikája következik a korábban elmondottakból. (3.2.1) esetében a jobb oldali kifejezés kiszámítása után törlődik az $x$ változó meglevő értéke, és helyébe íródik a jobb oldali kifejezés aktuális értéke. (3.2.2) végrehajtása közben az $A$ változó vagy a **true**, vagy a **false** logikai értéket veszi fel, aszerint, hogy a két előírt egyenlőség egyidejűleg teljesül-e vagy nem.

Magától értetődik, hogy egy értékadó utasítás csakis akkor hajtható végre, ha a jobb oldali kifejezés kiszámítható, azaz létezik aktuális értéke. Mivel az ALGOL-ban a változók csak értékadó utasításokkal kaphatnak értéket, következik, hogy mielőtt egy változó részt venne egy érték-

adó utasítás jobb oldali kifejezésében, előzőleg értékadó utasítás bal oldalán kellett szerepelnie.*

Mint tudjuk, aritmetikai változó kétféle van: egész és valós. Definiáljuk most a (3.1) értékadó utasítás szemantikáját arra az esetre, amikor $K$ aritmetikai kifejezés, és a bal oldali változó típusa különbözik a jobb oldali kifejezés aktuális értékének típusától! Az ALGOL mindkét lehetőséget megengedi: lehet a jobb oldali kifejezés egész, és a bal oldali változó valós, vagy fordítva.

Az első esetben a kapott aktuális érték egyszerűen átmegy egész ábrázolásúból valós ábrázolásúba. (Egész szám mindig ábrázolható valós módjára is.) A programozónak nem kell tudnia, mit hajt végre a gép, amikor az egész ábrázolású eredményt valós ábrázolásúvá alakítja át**, elég annyit tudnia, hogy egész típusú jobb oldali kifejezést kombinálhat valós típusú bal oldali változóval, és ezzel nem lép fel nemkívánatos effektus.

A fordított esetben természetesen valamilyen megállapodásra van szükség az átalakításra vonatkozólag. Az ALGOL szerzői a kézenfekvő „0.5-ön felül felfelé, azon alul lefelé" kerekítési szabályt választották. Tehát ha a jobb oldali kifejezés aktuális értékének törtrésze 1/2-nél nagyobb, akkor a gép felkerekít a legközelebbi egész számra, ellenkező esetben lekerekít. Képletben, ha $I$ egész típusú változó, $K$ valós aritmetikai kifejezés, akkor az $I := K$ értékadó utasítás egyenértékű a következővel:

$$I := entier(K + 0.5) \tag{3.3}$$

A (3.3) képletet, amely az ALGOL egyik fontos szemantikai szabálya, figyelembe kell venni programozás közben. E kerekítési szabály alkalmazásakor ügyelnünk kell arra, hogy a pontosan 1/2-re végződő számoknál másképpen megy végbe a kerekítés pozitív és másképpen negatív számokra. Pl. ha (3.3)-ban $K = +2.5$, akkor az $I$ változó

* Gépi reprezentánsok esetében ez nem áll fenn, mert az értékadó utasításokon kívül pl. beviteli utasítások is módosíthatják egy változó értékét.

** A gépi kódban való programozás szakkifejezésével élve, eközben a gép ún. normalizálást végez.

értéke 3 lesz, ha $K = -2.5$, akkor $-2$. Bináris gépeken a pontosan 1/2-re végződés elég gyakran előfordulhat!

Az eddigiekben két, egymásnak ellentmondó kerekítési szabályt ismertünk meg. Az elsőt az egészek osztásával kapcsolatban vezettük be [II. fejezet 4. p. (2.3) képlet], a második az értékadó utasítások (3.3) kerekítési szabálya. A kétféle kerekítés egyik következménye, hogy az

$$i := j/k \tag{3.4.1}$$

és

$$i := j \div k \tag{3.4.2}$$

értékadó utasítások bizonyos esetekben eltérő hatásúak (*i*, *j* és *k* egész típusú változók). Legyen például *j* aktuális értéke 20 és *k* aktuális értéke 7. Ebben az esetben az *i* változó a (3.4.1) utasítás hatására 3, a (3.4.2) utasítás hatására pedig 2 értékű lesz.

Az ALGOL nyelvben egy értékadó utasításnak nemcsak egy, hanem akárhány bal oldali változója lehet. Például

$$x := y := z := p + q \times x - y \tag{3.5}$$

Az ilyen utasításban az egyes bal oldali változók azonosítóit az értékadás jelével választjuk el egymástól, amint azt a (3.5) példán is láthatjuk. Az utolsó értékadási jel után következik a jobb oldali kifejezés.

A (3.5) típusú utasítások szemantikája a következő: a jobb oldali kifejezés kiszámításával kapott aktuális értéket át kell adni valamennyi bal oldali változónak. Fontos tudnunk, hogy a jobb oldali kifejezés az értékadó utasítás végrehajtása során csak egyszer számítódik ki.

A (3.5) típusú értékadó utasítások alkalmazása során be kell tartanunk még egy szintaktikus szabályt. Nevezetesen, a bal oldali változóknak mind azonos típusúaknak kell lenniük: vagy mindegyik egész, vagy mindegyik valós, vagy mindegyik logikai. Más szóval, a bal oldali változók listáján nem szabad összekeverni az egész és a valós típusú változókat.

Az ALGOL-utasításoknak egymástól való elkülönítésére külön alapjel van bevezetve, a pontosvessző.* Az összetartozó programrészek egyes utasításait ez választja el egymástól.

Tegyük fel, hogy ki kell számítanunk az alábbi aritmetikai kifejezést:

$$\frac{\sqrt{x^2+y^3+z^2+ln(b-a)}}{(x^2+y^3+z^2)\times(b-a)+Q}, \tag{3.6}$$

és az eredményt az $y$ változó kapja aktuális értékként. Elintézhetnénk a feladatot egyetlen értékadó utasítással is:

$$y:=sqrt(x\uparrow 2+y\uparrow 3+z\uparrow 2+ln(b-a))/((x\uparrow 2+y\uparrow 3+z\uparrow 2)\times (b-a)+Q) \tag{3.7}$$

A (3.7) értékadó utasítás azonban nem gazdaságos, mert a benne szereplő ismétlődő részkifejezéseket a gép annyiszor számítaná ki, ahányszor leírtuk azokat. Ráadásul, a fordítóprogram ugyanazon részkifejezések ismétlődő kiszámolására külön-külön programrészeket generálna a célprogramban, és így a célprogram nemcsak időben nem lenne gazdaságos, hanem memóriafelhasználás szempontjából sem. Éppen ezért a (3.6)-hoz hasonló kifejezéseket több lépésben célszerű számolni:

$$M1:=x\uparrow 2+y\uparrow 3+z\uparrow 2;\quad M2:=b-a;$$
$$y:=sqrt(M1+ln(M2))/(M1\times M2+Q); \tag{3.8}$$

Itt $M1$ és $M2$ segédváltozók (munkaváltozók), amelyek a számítás egyes részeredményeit tartalmazzák. A (3.8) program a (3.7) utasítással szemben kettővel több változót tartalmaz ($M1$ és $M2$), viszont a célprogram minden szempontból gazdaságosabb lesz.

Egy másik példa. Írjunk ALGOL-programot kétismeretlenes lineáris algebrai egyenletrendszerek megoldására. Legyen a megoldandó egyenletrendszer a következő:

$$\begin{aligned} a_{11}x_1+a_{12}x_2 &= b_1\\ a_{21}x_1+a_{22}x_2 &= b_2. \end{aligned} \tag{3.9}$$

* Hasonló hatása van még két másik alapjelnek is: az **end**-nek és az **else**-nek. Ezekkel később foglalkozunk.

Feltesszük, hogy a

$$D = a_{11}a_{22} - a_{12}a_{21} \tag{3.10}$$

kifejezés, az egyenletrendszer determinánsa, nem zérus. Ebben az esetben a megoldásokat az

$$x_1 = \frac{b_1a_{22} - b_2a_{12}}{D}$$

és

$$x_2 = \frac{a_{11}b_2 \div a_{21}b_1}{D} \tag{3.11}$$

képletek állítják elő. A (3.10) és (3.11) képleteket kell átírnunk ALGOL-programmá. Az egyenletrendszer együtthatóit az *a*11, *a*12, *a*21 és *a*22 azonosítókkal látjuk el (az index süllyesztése nincs megengedve), az ismeretlenek azonosítói *x*1 és *x*2. Az eddig tanultak alapján most már könnyen felírhatjuk a kívánt ALGOL-programot:

```
D:=a11 × a22 − a12 × a21;
x1:=(b1 × a22 − b2 × a12)/D;
x2:=(a11 × b2 − a21 × b1)/D
```

Az egyenletrendszer determinánsára vonatkozó megkötés programunk szempontjából azt jelenti, hogy a *D* változó aktuális értékének zérustól különbözőnek kell lennie, ellenkező esetben a program végrehajtása közben a gép 0-val való osztás miatt megáll. Megjegyezzük, hogy ha egy programban a változókra vonatkozó előzetes feltevéseink valamely oknál fogva nem teljesülnek, akkor a program mindig csak szemantikusan válik értelmetlenné (szintaktikusan sohasem), és ez a kellemetlenebb eset, mert a baj csak a célprogram működése közben derül ki.

Egy további megjegyzés.

Tekintsük a következő ALGOL-utasításcsoportot:

$$x:=p+q+r;\quad y:=p+q+r;\quad z:=p+q+r; \tag{3.12}$$

Legyenek a *p*, *q*, *r* változók aktuális értékei egyaránt 3.0-val egyenlők. A (3.12) program az *x*, *y*, *z* változóknak

egyformán a 9.0 értéket adja, és helyettesíthető egyetlen értékadó utasítással:

$$x:=y:=z:=p+q+r; \tag{3.13}$$

Több azonos jobb oldali kifejezéssel bíró értékadó utasítást azonban nem minden esetben szabad eggyé összevonni. Tekintsük pl. a következő utasításcsoportot:

$$x:=x+y+z; \quad y:=x+y+z; \quad z:=x+y+z; \tag{3.14}$$

Legyenek $x$, $y$, $z$ kezdeti aktuális értékei ismét 3.0-val egyenlők! (3.14) első utasítása 9.0-t ad értékként $x$-nek. A második utasítás már $x$ megváltozott aktuális értékével kerül végrehajtásra, és $y$-nak 15.0 értéket ad. A harmadik utasítás végrehajtása után $z$ értéke 27.0 lesz. Ezzel szemben az

$$x:=y:=z:=x+y+z \tag{3.15}$$

utasítás mindhárom változónak egyaránt 9.0 értéket ad, mivel a jobb oldali kifejezés csak egyszer kerül kiszámításra.

## 3. Címkék és vezérlésátadó utasítások

A címkék fogalmát az I. fejezet 3. p-ban ismertük meg. Emlékeztetünk rá, hogy a címkék az utasítások megnevezésére szolgálnak, és a vezérlésátadó utasításokkal kapcsolatban van fontos szerepük.

Az ALGOL 60 nyelvben címkeként tetszőleges azonosítót vagy előjel nélküli egész számot használhatunk. A címkékkel kapcsolatban bevezetünk egy új alapjelet, a kettőspontot. Ez választja el a címkét az utasítás többi részétől. Bármely ALGOL-utasítás viselhet akárhány címkét. Ha egy utasításnak több címkéje van, akkor ezeket a címkéket is kettőspont választja el egymástól. Például az

$$L2:Ro:3:x:=k+2\times i$$

utasításban $L2$, $Ro$ és 3 címkék; $L2$ és $Ro$ azonosító, 3 pedig előjeltelen egész.

A legtöbb ALGOL gépi reprezentáns megtiltja előjeltelen egészek címkeként való alkalmazását. Mi a továbbiakban kizárólag azonosítókat fogunk címkeként használni; egyébként bármely előjeltelen egész szám azonosítóvá változik, ha tetszőleges betűt írunk eléje, ezért a számcímkék kizárása lényegtelen megszorítás.

Az ALGOL-ban a vezérlésátadó utasítás szerepét a **go to** utasítás tölti be. A **go to** (menj . . .-hoz) alapszópár a közte levő szóközzel együtt egyetlen alapjelnek tekintendő, tehát a **go** és a **to** nem külön-külön alapjelek.*

A **go to** utasítás legegyszerűbb változatában a **go to** alapjel után valamely címke áll:**

**go to** *L* (3.16)

A (3.16) utasítás hatására a vezérlés megkeresi azt az ALGOL-utasítást, amely az *L* címkét viseli, és a program végrehajtását ennél az utasításnál folytatja. Bizonyos feltételek teljesülése esetén lehet egy programban több, azonos módon felcímkézett utasítás is; erről az esetről egy külön kiválasztási szabály rendelkezik, amelyet szintén az V. fejezetben, a címkék hatásköréről szóló pontban (5. p.) fogunk megismerni.

*Példa*

Legyen egy ALGOL-programban a következő programrészlet:***

$x:=(b-a)/(b+a)$;
**go to** *gamma*; $P: y:=0$; (3.17)
*gamma*: $z:=u-v$; {folytatás}

A (3.17) programrészlet első utasításának végrehajtása után a gép — a soron következő **go to** utasítás hatására — ki-

* Miután a **go to**-nak egybe-, illetve különírása állandó vita tárgya, egyes fordítóprogramok így is, úgy is elfogadják.

** Az általánosabb esetet a IV. fejezet 4. p.-ban tárgyaljuk.

*** Ezt, és a könyv ezen részének valamennyi programját egy-egy nagyobb program kiragadott részének tekintjük, mivel a közölt programok legalábbis abban az értelemben nem teljesek, hogy hiányzik belőlük a deklarációs rész. Itt és a következőkben a programok azon részeit, amelyeket nem írunk ki, de jelenlétüket valami módon érzékeltetni kívánjuk, kapcsos zárójelbe zárt szavakkal jelezzük, pl.: {folytatás}.

kerülli a *P* címkével ellátott utasítást, és közvetlenül a gamma címkéjű értékadó utasításra tér át.

A **go to** utasítások is viselhetnek címkét. Speciálisan, ha egy **go to** utasítás ugyanazt a címkét viseli, mint amelyre ugrat, akkor, ha egyszer rákerült erre az utasításra, ettől kezdve mindig ugyanezt fogja ismételni a végtelenségig. Ilyen utasítás a következő:

$$L:\textbf{go to } L;$$

Ezt a szituációt *végtelen ciklusnak* nevezzük. Szándékosan nem szokás előidézni, legalábbis ALGOL-programokban nem.* Annál gyakrabban előfordul programozási hiba következtében, hogy a gép olyan programhurokba kerül, amelyből nincs kijárat. Egyes gépi reprezentánsok ún. „nyomkövetési" lehetőségeket biztosítanak, amelyek megkönnyítik a végtelen programhurkok programbeli helyének és létrejöttük okának felderítését.

## 4. Feltételes utasítások és feltételes kifejezések

Az alábbiakban szükségünk lesz három újabb szóalapjelre. Ezek a következők: **if** (ha), **then** (akkor) és **else** (egyébként).

Az **if** és a **then** alapjelek segítségével definiálunk egy fontos szintaktikus egységet, amelynek az ALGOL 60 nyelvben rendkívül nagy szerepe van. Ez a szintaktikus egység a *logikai feltétel*. Logikai feltételt akkor kapunk, ha az **if** és a **then** alapjelek közé egy logikai kifejezést írunk. A logikai feltétel általános alakja tehát a következő:

$$\textbf{if } L \textbf{ then} \tag{3.18}$$

ahol $L$ tetszőleges logikai kifejezés.

Összhangban a szokásos szóhasználattal, egy logikai feltételről akkor mondjuk, hogy teljesül, ha a benne sze-

* Bizonyos gépek belső kódjában nincs megállási utasítás, és ezért ezeken a gépeken a megállást szándékosan előidézett végtelen ciklussal imitálják.

replő $L$ logikai kifejezés aktuális értéke **true**, vagyis igaz, és azt mondjuk, hogy nem teljesül, ha $L$ aktuális értéke **false**, vagyis hamis.

A logikai feltételek első alkalmazásaként megismerjük most a *feltételes utasítás* fogalmát. A feltételes utasításnak két alaptípusa van, mégpedig:

**if** $L$ **then** $U1$ **else** $U2$ (3.19.1)

és

**if** $L$ **then** $U$ (3.19.2)

Itt $L$ tetszőleges logikai kifejezés, $U1$, $U2$ és $U$ pedig ALGOL-utasítások.

A (3.19.1) utasítás végrehajtási szabálya a következő:

1. Megvizsgálandó, hogy a feltételes utasításban szereplő logikai feltétel teljesül-e.

2. Ha a feltétel teljesül, akkor végrehajtandó az $U1$ utasítás.

3. Ha a feltétel nem teljesül, akkor végrehajtandó az $U2$ utasítás.

4. A logikai feltételnek megfelelő $U1$, ill. $U2$ utasítás végrehajtása után, a feltételes utasítást a felírás sorrendjében követő ALGOL-utasításra kell áttérni, hacsak a végrehajtott ($U1$, ill. $U2$) utasítás ettől eltérően nem rendelkezik.

*Megjegyzés:* Egy (3.19.1) típusú utasítás végrehajtása során az $U1$ és $U2$ utasítások közül mindig csakis az egyiket kell végrehajtani. Az ALGOL nyelv szabályai megengedik, hogy a másik utasítás az adott pillanatban szemantikusan értelmetlen legyen, például értékkel nem bíró változó szerepeljen benne, vagy az őt felépítő kifejezések ne legyenek kiszámíthatók (zérus nevező, gyök alatti negatív mennyiség stb. miatt). Az utasításnak azonban szintaktikusan rendben levőnek kell lennie. Egy szemantikai rendellenesség „rendbejöhet", mire a gépen a kérdéses utasítás végrehajtására sor kerül, ha ugyan egyáltalán valaha is sor kerül rá, de egy szintaktikus hiba nem jön rendbe, és a fordítóprogram le sem fordítja az ALGOL-programot, ha az szintaktikusan hibás.

*Példák*

**1.** Tekintsük az alábbi ALGOL-programrészletet:

$x:=2+y;$
**if** $A-x<_{10}-3$ **then** $X:=X+x$ **else** $A:=A+t;$
$t:=2\times t;$ {folytatás} (3.20)

A számítás menete a következő. A gép először végrehajtja a program első értékadó utasítását. Ezután elérkezik a feltételes utasításhoz. Megvizsgálja a logikai feltételt, és ennek alapján választ a **then** utáni, és az **else** utáni utasítások között. Ha például az első utasítás végrehajtása után $A$ aktuális értéke 5.7461, és $x$ aktuális értéke 5.7468, akkor a logikai feltétel teljesül, és ezért a **then** utáni utasítás kerül végrehajtásra. Ez ismét értékadó utasítás; a gép végrehajtja, és mivel eltérő utasítást nem kapott, áttér a $t:=2\times t$ utasításra.

Itt rögtön megjegyezzük, hogy az **else** alapjel az ALGOL-ban végjel értékű alapjel, és ezért külön pontosvesszőre az **else** előtt nincs szükség, sőt, **else** elé nem is szabad pontosvesszőt írni.

**2.** Vizsgáljuk a következő ALGOL-programot:

$L:x:=x/2;$ **if** $x\geq_{10}-4$ **then go to** $L$ **else** $t:=x;$
$P:=t\uparrow 2;$ {folytatás} (3.21)

A (3.21) program szemantikája a következő: a gép kiindul valamilyen $x$ kezdőértékből, és elkezdi ezt az értéket felezni. Mindaddig újra és újra felezi, amíg az $_{10}-4$-nél kisebbé nem válik. Amikor ez bekövetkezett, az utolsó felezés után kapott értéket beírja a $t$ változóba, négyzetét pedig $P$-be. A **then** utáni **go to** segítségével feltétlen vezérlésátadást generáltunk, amely mindig újra és újra visszairányítja a vezérlést az $L$ címkével ellátott értékadó utasításra, mindaddig, amíg az $x$ mennyiség elegendően kicsinynyé nem vált, és csak ekkor engedélyezi a $t:=x$ utasítás végrehajtását. Ezzel ún. programhurkot (ciklust) állítottunk elő.

Áttérünk most a (3.19.2) típusú feltételes utasítás tárgyalására. Ez szemantikailag még egyszerűbb, mint (3.19.1). A szabályok a következők:

1. Megvizsgálandó a logikai feltétel teljesülése.
2. Ha teljesül, végrehajtandó az *U* utasítás.
3. Ha nem teljesül, át kell térni a soron következő utasításra.
4. Amennyiben az *U* utasítás eltérően nem rendelkezik, annak végrehajtása után szintén a soron következő utasítással kell folytatni a program végrehajtását.

*Példák*

1. Tanulságos összehasonlításként először ismét a (3.21) algoritmust vizsgáljuk, ezúttal a (3.19.2) típusú feltételes utasítás felhasználásával.

```
L:x:=x/2;  if x ≧10 − 4 then go to L;                    (3.22)
t:=x;  P:=t↑2;  {folytatás}
```

A (3.22) program szó szerint ugyanazt hajtatja végre a géppel, mint a (3.21) változat. Formailag azonban különbözik a kettő, mégpedig annyiban, hogy (3.22)-ben az **else** helyett pontosvessző áll. A (3.19.2) utasítások szemantikájáról mondottakkal összhangban, a gép mindaddig a **then** utáni utasítást hajtja végre, amíg az $x \geqq_{10} -4$ logikai feltétel teljesül. Amint ennek a feltételnek aktuális értéke **false**-ra változik, a gép, átugorva a **go to** *L* utasítást, áttér a soron következő utasításra, amely éppen az az utasítás, amely a (3.21) változatban az **else** után áll.

A (3.21) és (3.22) programok összevetése alapján azonban nem szabad arra a téves következtetésre jutni, hogy az **else** mindig fölösleges, és annak pontosvesszővel való helyettesítése után mindig azonos jelentésű programot kapunk. A (3.19.1) utasítás szemantikájának legjellegzetesebb vonása a benne rejlő *kizáró* alternatíva („*U*1 és *U*2 nem, vagy *U*2 és *U*1 nem"). Ha az *U*1 utasítás máshová adja a vezérlést, és nem használjuk ki a feltételes utasításnak azt a tulajdonságát, hogy automatikusan biztosítja a feltétel teljesülése esetén az *U*2 utasítás kikerülését, akkor az **else** rendszerint fölöslegessé válik. Ha azonban programunk

olyan, hogy az $U1$ utasítás végrehajtása után a természetes sorrend szerint haladunk tovább, tehát a feltételes utasítást követő utasításra térünk át, akkor olyan programra jutunk, amelyben teljesen ki vannak használva a (3.19.1) utasítástípus által nyújtott lehetőségek.

2. Írjunk ALGOL-programot a négyzetgyökvonásra! Amint ez a numerikus analízis elemeiből ismeretes, négyzetgyök vonására az alábbi iterációs formula alkalmazható:

$$x_{k+1} = 0.5\left(x_k + \frac{A}{x_k}\right). \tag{3.23}$$

Ha a (3.23) képletet értékadó utasítássá írjuk át, elhagyva az iterációs indexeket, akkor lényegében előttünk áll a négyzetgyökvonás ALGOL-programja, csak ki kell azt egészítenünk néhány vezérlő utasítással. Ezt fogjuk most megtenni. Az iteráció kezdőértékéül válasszunk pl. $x_0 = 1$-et!

A program a következő:

```
if A=0 then go to EXIT; x:=1;
L: xe:=x;   x:=0.5×(x+A/x);
if abs(x-xe)≧10-4×x then go to L;   go to out;
EXIT: x:=0;   out: {folytatás}
```

Az első utasítás megvizsgálja, hogy a gyökvonandó zérus-e vagy sem. Az iterációs program ugyanis $A=0$ esetében nem működne jól, és ezért azt, hogy ebben az esetben is helyes eredményt kapjunk, külön utasítással kell biztosítanunk. A következő utasítás beállítja a kezdőértéket, és ezután kezdődik a tulajdonképpeni iterációs ciklus. Minden iterációs lépésben a gyöknek két egymás utáni közelítését használjuk fel, hogy összehasonlíthassuk azok eltérését. E célból bevezettük az *xe* segédváltozót, amely az előző iteráltat tárolja. Az *abs* $(x-xe)\geqq_{10}-4\times x$ feltétel önkényes, és konkrét esetekben esetleg mással kell helyettesíteni.

Az ALGOL szintaktikus szabályai megengedik, hogy a (3.19.1) feltételes utasításban $U2$ maga is feltételes utasítás legyen. Ebben az esetben az **else** alapjel után újabb **if** következik. Ha az **else** utáni feltételes utasítás ismét (3.19.1) típusú, akkor az abban szereplő $U2$ megint lehet

akár (3.19.1), akár (3.19.2) típusú feltételes utasítás. A második **else** után következhet egy harmadik **if** stb. Ilyen módon tetszőleges hosszúságú, ún. *láncolt feltételes utasítások* jöhetnek létre. A lánc akkor ér véget, amikor egy **else** után feltétlen, vagy (3.19.2) típusú feltételes utasítás következik. Az alábbi jelsorozatok három tagból álló feltételláncokat mutatnak be:

**if** *L*1 **then** *U*1 **else if** *L*2 **then** *U*2 **else if** *L*3 **then** *U*3 **else** *U*4; *V*; (3.24.1)

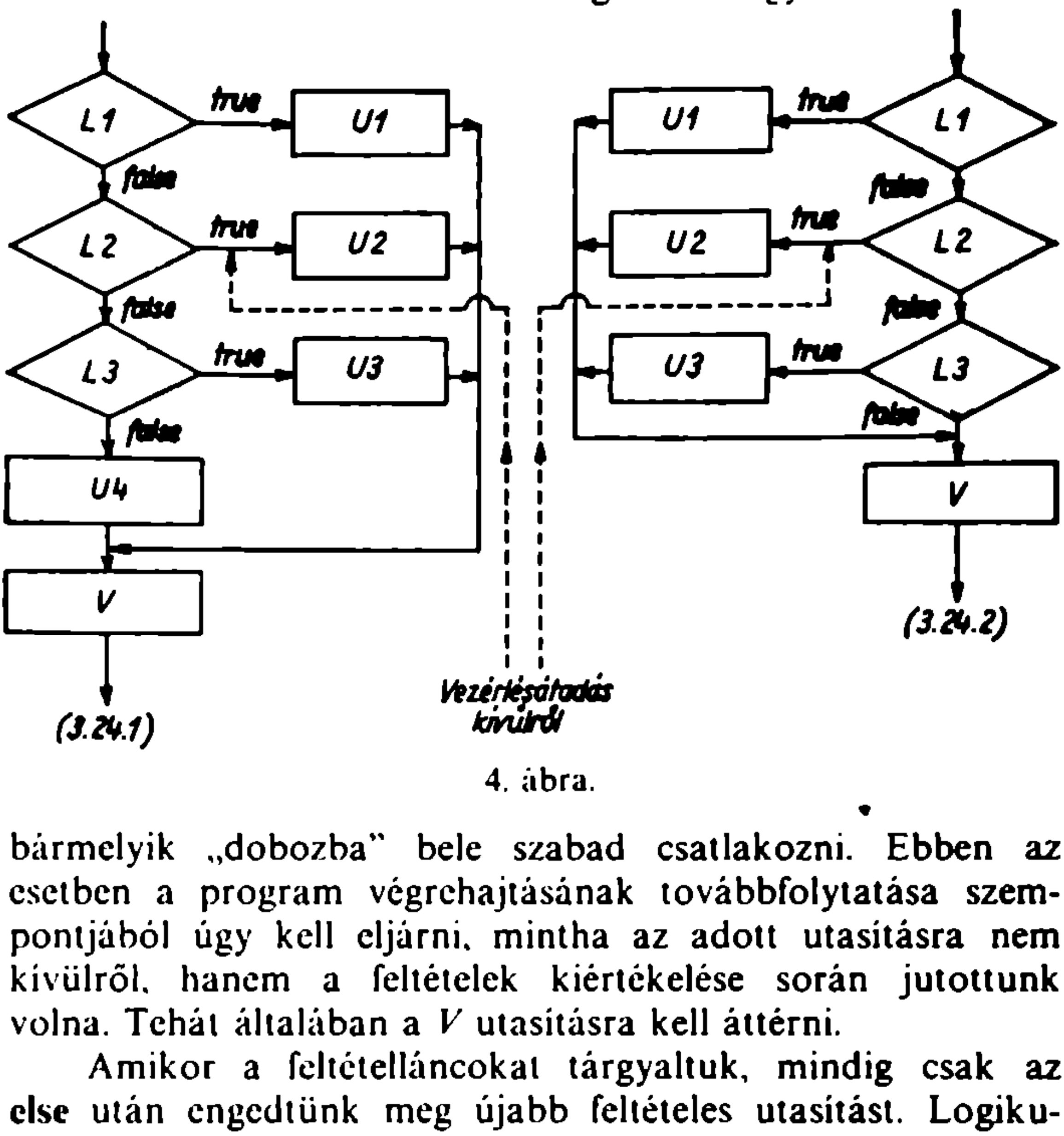
**if** *L*1 **then** *U*1 **else if** *L*2 **then** *U*2 **else if** *L*3 **then** *U*3; *V*; (3.24.2)

A (3.24.1) utasításnál feltesszük, hogy *U*4 feltétlen utasítás, és a lánc valóban véget ér a harmadik tag után. A (3.24.2) utasításlánc harmadik tagja (3.19.2) típusú, és így a lánc továbbfolytatására nincs lehetőség. Mindkét esetben *V* a következő utasítást jelenti.

Mindkét utasításláncra vonatkozó végrehajtási szabályok a már ismert szabályok rekurzív alkalmazásával nyerhetők. Tekintsünk egy (3.24.1) típusú utasításláncot, és tegyük fel általánosságban, hogy az *m* tagból áll (példáinkban $m=3$). Tudjuk, hogy a logikai feltétel nem teljesülése esetén az **else** utáni utasításra kell áttérni. Feltétellánc esetében itt újabb logikai feltételre jutunk, tehát megint alkalmaznunk kell a választási szabályt. Könnyű meggondolni, hogy végül is a következőre jutunk: egy (3.24.1) típusú feltételes utasítás hatásában egyenértékű az *U*1, *U*2, ... *Um* utasítások közül az *i*-edikkel, ha az első $i-1$ logikai feltétel egyike sem teljesül, de az *i*-edik teljesül; és az utolsó **else** utáni ($m+1$-edik) utasítással, ha az *m* számú logikai feltételből egy sem teljesül. A (3.24.2) utasításoknál teljesen hasonló a helyzet, csak annyi a különbség, hogy ha valamennyi logikai feltétel **false** értékű, akkor ez a fajta utasítás teljesen hatástalan. A logikai feltételek által kiválasztott utasítás után — amennyiben az eltérően nem rendelkezik — a felírás sorrendjében soron következő utasítást kell végrehajtani.

A fentebb elmondottakat a 4. ábrán feltüntetett blokkdiagramokkal szemléltethetjük.

Hozzá tartozik a feltételes utasítások szemantikájához a címkék kérdése is. Az $U1$, $U2$, ... utasítások bármelyike viselhet címkét, akár a (3.24.1), akár a (3.24.2) utasításokban. Szabad erre a címkére ráugratni, vagyis az ábrákon




4. ábra.

bármelyik „dobozba" bele szabad csatlakozni. Ebben az esetben a program végrehajtásának továbbfolytatása szempontjából úgy kell eljárni, mintha az adott utasításra nem kívülről, hanem a feltételek kiértékelése során jutottunk volna. Tehát általában a $V$ utasításra kell áttérni.

Amikor a feltételláncokat tárgyaltuk, mindig csak az **else** után engedtünk meg újabb feltételes utasítást. Logikusan azt várná az ember, hogy ha az **else** után szabad újabb feltételes utasítást kezdeni, akkor szabad a **then** után is. Ez azonban nem így van. Az ALGOL-ban **then** *után* **if**-*et semmilyen körülmények között nem szabad írni!*

Bemutatunk most egy példát ennek indoklására. Ha meg lenne engedve, hogy a (3.19.1) vagy (3.19.2) utasításokban a **then** után is írhassunk **if**-et, akkor szintaktikusan helyes lenne az alábbi ALGOL-utasítás:

$$\textbf{if } i = k \textbf{ then if } p = q \textbf{ then } x := y \textbf{ else } u := v \qquad (3.25)$$

Azonban a (3.25) utasítás nem lenne szemantikailag értelmezhető, mert nem lehetne megállapítani, hogy az egyetlen **else** a két **then** közül melyikhez tartozik. (3.25)-öt ui. lehetne úgy is értelmezni, mint egy (3.19.1) típusú utasítást, amelyben $U1$ helyébe egy (3.19.2) utasítás van behelyettesítve, vagy értelmezhetnénk úgy is, mint egy (3.19.2) típusú utasítást, amelyben $U$ helyébe egy (3.19.1) típusú utasítás van behelyettesítve. [Javasoljuk az Olvasónak, mindkét lehetséges értelmezés mellett gondolja meg, hogyan működne (3.25), ha $i=3$, $k=4$ és $v=2.5$!]

A **then** utáni **if** tilalmának megkerülésére több lehetőség kínálkozik. (3.19.1) típusú utasítások esetén, ha $U1$ feltételes utasítás lenne, de $U2$ nem, akkor egyszerűen a feltételt ellenkezőjére fordítjuk, és kicseréljük $U1$-et $U2$-vel. Tehát

**if** $L$ **then** $U1$ **else** $U2$

helyett írhatjuk:

**if** $\neg L$ **then** $U2$ **else** $U1$

és ezzel visszavezettük egy (3.24.1) típusú utasításra.

Néha célravezető a logikai műveletek alkalmazása. Tegyük fel például, hogy a következő utasítás felírására volna szükség:

**if** $x=1$ **then if** $y=2$ **then** $x:=2\times y$

Itt a **then** utáni **if** eltüntethető a logikai szorzás műveletének segítségével:

**if** $x=1 \wedge y=2$ **then** $x:=2\times y$

Ha a fentiekhez hasonló fogások nem vezetnek eredményre, akkor az ún. utasítás-zárójelek alkalmazásához kell folyamodnunk. Ezekről a következő pontban lesz szó.

Bevezetünk most még egy fogalmat, amelyet az ALGOL-ban elég gyakran használnak. *Üres utasításnak* nevezünk egy olyan ALGOL-utasítást, amely nem áll másból, mint egy magában álló végjel értékű jelből, amilyen pl. egy magában álló pontosvessző. Az üres utasítás teljesen hatástalan. Viselhet az üres utasítás címkét is, ebben az esetben a végjel közvetlenül a címkét lezáró kettőspont után áll. Példa az üres utasításra:

*alfa* : ; (3.26)

Később látni fogjuk, hogy hasonló konstrukciót főként az **end** alapjellel kapcsolatban szoktunk alkalmazni [lásd például a IV. fejezetben a Simpson-integráció programját, (4.19)-et].

Az üres utasításnak a szabatos beszédmód szempontjából is van jelentősége. Ha egy utasításról azt akarjuk mondani, hogy bizonyos feltételek teljesülése esetén hatástalan, ezt úgy is kifejezhetjük, hogy „hatásában egyenértékű egy üres utasítással". Így például a (3.19.2) utasítás hatása egyenértékű az üres utasítással, ha a benne szereplő logikai feltétel nem teljesül. Az Olvasóra bízzuk a (3.24.2) utasítás szemantikájának átfogalmazását, a bevezetett terminus segítségével.

A fejezet elején megismert „logikai feltétel" szintaktikus kategóriájának segítségével a feltételes utasításokon kívül. más jelsorozatokat is képezhetünk. Az alábbiakban megismerjük a *feltételes kifejezés* fogalmát. A feltételes kifejezések lehetnek aritmetikai és logikai típusúak. A II. fejezetben megismert feltétlen kifejezésekhez hasonlóan, a feltételes kifejezések is egy szám-, illetve logikai értéket határoznak meg, amely a kifejezésben szereplő változók aktuális értékeinek függvénye. Következésképpen a feltételes kifejezések, akárcsak a feltétlenek, nem utasítások, hanem csak utasítások alkatrészei. Azért tárgyaljuk őket mégis az utasítások között, mert szemantikájuk igen sok tekintetben hasonló a feltételes utasításokéhoz.

Szintatikus szempontból, a feltételes aritmetikai kifejezés általános alakja a következő:

**if** *L* **then** *A*1 **else** *A*2 (3.27)

Itt *L* tetszőleges logikai kifejezés, *A*1 és *A*2 aritmetikai kifejezések.

Példák feltételes kifejezésekre:

**if** $A \wedge B$ **then** $x - y$ **else** $a\uparrow 2 - b\uparrow 2$ (3.28.1)

**if** $i \neq j$ **then** $a \times b$ **else** $1/a$ (3.28.2)

**if** $A \equiv \neg B \vee C$ **then** *sqrt* $(1 - x\uparrow 2)$ **else** 0 (3.28.3)

A feltételes aritmetikai kifejezések kiszámítási szabálya messzemenő analógiát mutat a feltételes utasítások végrehajtási szabályával. Eszerint:

1. A kifejezés kiszámítását a logikai feltétel kiértékelésével kell kezdeni.

2. Ha a logikai feltétel teljesül, akkor kiszámítandó a **then** utáni *A*1 aritmetikai kifejezés aktuális értéke, és ez az érték lesz a feltételes kifejezés aktuális értéke is.

3. Ha a logikai feltétel nem teljesül, akkor kiszámítandó az **else** utáni *A*2 aritmetikai kifejezés aktuális értéke, és ez az érték adja a feltételes kifejezés értékét.

*Megjegyzés:* A (3.27) feltételes aritmetikai kifejezés kiértékelése közben az *A*1 és *A*2 aritmetikai kifejezések közül mindig csakis az egyiket kell kiszámítani. Ugyanebben az időpillanatban a másik kifejezés szemantikailag értelmetlen is lehet.

*Példák*

1. Számítsuk ki például a (3.28.1) kifejezés aktuális értékét, ha $A =$ **true**, $B =$ **false**, $a = 3$ és $b = 4$! A logikai feltétel ebben az esetben nem teljesül, és így az **else** utáni kifejezés aktuális értékét kell kiszámítani. Ez $-7$-tel egyenlő; ha feltesszük, hogy $a$ és $b$ közül legalább az egyik valós típusú változó, akkor az eredmény is valós típusú lesz. A feltételes kifejezésben szereplő $x$ és $y$ változók aktuális értéke jelen esetben közömbös a kifejezés aktuális értéke szempontjából, az sem baj, ha egyáltalán nincs is értékük.

2. Írjunk olyan értékadó utasítást, amely 1-et ír az $y$ változóba, ha $x$ kívül van a $[-1, +1]$ intervallumon, és $1 + x^2$-et, ha $x$ az említett intervallum belsejében vagy határán van!

A kívánt értékadó utasítás a következő lehet:

$$y := \textbf{if} -1 \leqq x \wedge x \leqq +1 \textbf{ then } 1 + x\uparrow 2 \textbf{ else } 1$$

3. Írjunk olyan értékadó utasítást, amely a $p$ változóba elhelyezi az $m$ és $n$ változók közül azt, amelyiknek az abszolút értéke nagyobb!

Megfelel a következő utasítás:

$$p := \textbf{if } abs\,(m) > abs\,(n) \textbf{ then } m \textbf{ else } n$$

A feltételes aritmetikai kifejezések főként — de nem kizárólag — értékadó utasítások jobb oldali kifejezéseiként fordulnak elő. A feltételes utasításokkal ellentétben, mindig szerepel bennük az **else** alapjel. A (3.19.2) típusú, **else** nélküli feltételes utasításoknak sem a feltételes aritmetikai, sem a feltételes logikai kifejezések körében nincs megfelelője.

A feltételes utasításokhoz hasonlóan, feltételes kifejezésekből is képezhetünk láncokat. Feltételláncok úgy jönnek létre, hogy egy feltételes kifejezés **else** utáni kifejezése maga is feltételes kifejezés. Ilyenkor az **else** után újabb **if** következik. Tetszőleges hosszúságú láncok megengedettek. A következő jelsorozat három tagból álló láncot mutat be:

**if** $L1$ **then** $A1$ **else if** $L2$ **then** $A2$ **else if** $L3$ **then** $A3$ **else** $A4$ (3.29)

A (3.29) típusú feltételes kifejezések kiszámítási szabálya az ismert szabály rekurzív alkalmazásával nyerhető. E szabály szerint, egy (3.29) típusú feltételes kifejezés aktuális értéke az $Ai$ aritmetikai kifejezés aktuális értékével egyenlő, ha a láncban szereplő első $i-1$ logikai feltétel nem teljesül, de az $i$-edik teljesül, és a legutolsó **else** utáni aritmetikai kifejezés aktuális értékével egyenlő, ha a logikai feltételek egyike sem teljesül.

Feltételláncok alkalmazásaként oldjuk meg a következő feladatot:

Legyenek $a$, $b$, $c$, $d$, $e$, $f$, $x$ és $m$ valós típusú változók. Feltesszük, hogy az $x$ változó aktuális értékére az

$$a \leqq x \leqq b, \tag{3.30.1}$$

$$c \leqq x \leqq d \tag{3.30.2}$$

és

$$e \leqq x \leqq f \tag{3.30.3}$$

egyenlőtlenségek közül egy és csakis egy teljesül. Írjunk ALGOL értékadó utasítást, amely az $m$ változóba helyezi el $a$, $c$, ill. $e$ értékét aszerint, hogy a három egyenlőtlenség

közül rendre az első, a második, ill. a harmadik teljesül-e. Megoldásként alkalmas a következő utasítás:

$m :=$ **if** $a \leqq x \wedge x \leqq b$ **then** $a$ **else if** $c \leqq x \wedge x \leqq d$ **then** $c$ **else** $e$

A harmadik egyenlőtlenség teljesülését már nem kell vizsgálnunk, mert ha az első kettő nem teljesül, akkor a feladat feltevéseinek értelmében a harmadiknak biztosan teljesülnie kell.

A feltételes kifejezések esetében hasonlóan a feltételes utasításokhoz, szintén fennáll a **then** utáni **if** tilalma, tehát csak az **else** után állhat ismét feltételes kifejezés. A **then** utáni **if** a feltételes utasításokra ismertetett módon itt is elhárítható, azonban a problémát teljes általánosságban megoldja az a fontos szemantikai szabály, amely szerint bármely feltételes kifejezés elsődleges kifejezésnek számít, ha zárójelbe zárjuk. Ez azt jelenti, hogy ha a feltételes kifejezést, amelyet a **then** után kívánunk elhelyezni, kerek zárójelbe zárjuk, ezáltal megszüntettük feltételes jellegét, tehát baj nélkül alkalmazhatjuk. Igaz: most már nem **if** következik a **then** után, hanem kezdőzárójel, amit már megenged a szintaxis.

Feltételes kifejezés nem lehet semmiféle kétparaméteres művelet operandusa. Szintaktikusan hibás például a következő jelsorozat:

$$a + \textbf{ if } x < y \textbf{ then } b \textbf{ else } c \tag{3.31}$$

Mivel azonban minden feltételes kifejezés elsődleges kifejezéssé válik, ha zárójelbe helyezzük, és elsődleges kifejezés már lehet operandus aritmetikai műveletben, a (3.31)-ben szereplő feltételes kifejezést zárójelbe zárva, már a szintaxissal összeférő jelsorozatot kapunk:

$$a + (\textbf{if } x < y \textbf{ then } b \textbf{ else } c) \tag{3.32}$$

Első pillanatban nem tűnik érthetőnek, hogy (3.31)-et miért nem lehet megengedni? (3.31) értelmezésével nem is lenne semmi baj. Azonban cseréljük fel (3.31) két tagját, és máris olyan jelsorozatot kapunk, amelynek szemantikája nem állapítható meg egyértelműen:

$$\textbf{if } x < y \textbf{ then } b \textbf{ else } c + a \tag{3.33}$$

(3.33)-at érthetnénk akár így:

**if** $x < y$ **then** $b$ **else** $(c + a)$

akár így:

**(if** $x < y$ **then** $b$ **else** $c) + a$

Azzal, hogy a feltételes kifejezést zárójelbe zárjuk, minden esetben egyértelmű jelsorozatot kapunk. Igaz, hogy ezáltal tiltottá válnak olyan jelsorozatok is, amelyeket lehetett volna értelmezni. Viszont ha ezt el akarnánk kerülni, a szintaxis bonyolultabb lenne, mert pontosan körül kellene határolni, hogy mely esetben engedhető meg operandusként feltételes kifejezés, és mely esetben nem.

A továbbiakban az aritmetikai kifejezés legáltalánosabb esetének a feltételes kifejezést tekintjük. Ez azt jelenti, hogy mindenütt, ahol aritmetikai kifejezések kerülnek szóba, beleértjük ezek közé a feltételes kifejezéseket is, hacsak ennek ellenkezőjét külön ki nem kötjük.

Nincs megengedve feltételes aritmetikai kifejezések alkalmazása relációk jobb és bal oldalán. A II. fejezet 7. pont elején, a reláció közölt definíciójában tehát lényeges szerepe van annak, hogy *relációk képzésénél feltétlen kifejezéseket kell relációjellel összekapcsolni.* Erre annak idején nem hivhattuk fel a figyelmet eléggé nyomatékosan, miután addig csak a feltétlen kifejezéseket ismertük. E tilalom egyébként szintén összefügg a **then** utáni **if** tilalmával, amint azt a feltételes logikai kifejezések tárgyalása során látni fogjuk.

Eddig két kizáró tilalmat ismertünk meg a feltételes aritmetikai kifejezésekre vonatkozólag: az egyik a feltételes kifejezések **then** utáni írására vonatkozik, a másik a relációkra. További korlátozást a szintaxis nem is tesz, tehát e két kivételtől eltekintve, a feltételes aritmetikai kifejezések korlátlanul alkalmazhatók.

A feltételes aritmetikai kifejezésekhez teljesen hasonlóak a feltételes logikai kifejezések, melyek egyetlen logikai értéket határoznak meg; ez az érték függ a kifejezést felépítő változók aktuális értékétől. A feltételes logikai kifejezés általános alakja a következő:

**if** *L* **then** *K*1 **else** *K*2 (3:34)

Itt *L*, *K*1 és *K*2 logikai kifejezések.

A (3.27) feltételes aritmetikai kifejezés szemantikájáról elmondottak — így különösen a három pontba foglalt kiszámítási szabály, a (3.29) típusú feltételláncok kiértékelési szabálya és a **then** utáni **if**-fel kapcsolatos tudnivalók — csaknem minden változtatás nélkül érvényesek itt is, így azok ismételt leírásától eltekintünk.

Példák a feltételes logikai kifejezésekre:

**if** $i = k$ **then** $A \wedge B \supset C \vee D$ **else** $A \wedge C \supset B \vee D$ (3.35.1)

**if** $(A \equiv B) \wedge \neg C$ **then** $x + p \times y > k \div l$ **else true** (3.35.2)

**if** $A \wedge B \wedge C$ **then** $\neg E \wedge F \vee G$ **else** $A \vee \neg B \vee C$ (3.35.3)

Számítsuk ki a (3.35.1) kifejezés aktuális értékét, ha $i = 3$, $k = 4$, $A = C =$ **true** és $B = D =$ **false**!

Miután $3 \neq 4$, a kifejezés aktuális értékének meghatározásához az **else** utáni kifejezés aktuális értékét kell meghatároznunk. A II. fejezet 8. p.-ban megismert szabályok szerint ez az érték **false**, és így (3.35.1) aktuális értéke is **false**.

Az Olvasóra bízzuk annak belátását, hogy $A = B =$ **true**; $C =$ **false**; $x = 3.6$; $p = 0.4$; $y = 1.2$; $k = 11$ és $l = 3$ aktuális értékek mellett (3.35.2) aktuális értéke **true**, továbbá hogy $A = B =$ **true** és $C =$ **false** esetén — függetlenül attól, hogy az *E*, *F* és *G* változóknak van-e értékük, és ha van, akkor mi ez az érték — (3.35.3) aktuális értéke szintén **true**.

A (3.34) kifejezésben *K*2 maga is lehet feltételes kifejezés. Ennek **else**-je után ismét következhet feltételes kifejezés s. í. t. Ez a konstrukció elvezet a (3.29) jelsorozattal kapcsolatban már tárgyalt feltételláncokhoz, melyekkel most nem foglalkozunk részletesen. Az ALGOL szintaxisa azonban azt is megengedi, hogy (3.34)-ben *L* legyen feltételes logikai kifejezés, és ezzel egy másfajta feltétellánchoz jutunk:

**if if** *L*1 **then** *K*1 **else** *M*1 **then** *K*2 **else** *M*2 (3.36)

A (3.36) jelsorozat aktuális értékének kiszámítását az ismert szabály rekurzív alkalmazásával kell végezni. Eszerint

a számítást a logikai feltétel kiértékelésével kell kezdeni. Az $L$ logikai feltétel helyébe itt ismét egy (3.34) típusú logikai kifejezés került. Ezt tehát megint csak az ismert algoritmus alapján kell kiszámítani. Az $L1$ logikai feltétel teljesülése, ill. nem teljesülése alapján választunk a $K1$ és $M1$ logikai kifejezések között. Kiértékeljük a megfelelőt, és ennek aktuális értéke lesz (3.36) feltételének aktuális értéke, tehát ezen érték alapján választunk a $K2$ és $M2$ kifejezések között. (3.36) aktuális értéke végeredményben $K2$ és $M2$ közül a megfelelőnek aktuális értéke.

(3.36)-ban $L1$ ismét lehet feltételes kifejezés. Ilyenkor három **if** áll egymás után. Ilyen kifejezés kiszámításánál algoritmusunkat háromszor kell rekurzíve alkalmazni. Az ALGOL 60 nyelv megengedi a feltételek akárhányszoros egymásba skatulyázását. Megjegyezzük, hogy egymásba skatulyázott feltételek nemcsak logikai, hanem aritmetikai kifejezésekben is előfordulhatnak; (3.27)-ben $L$ lehet feltételes logikai kifejezés. A feltételes kifejezések szintaxisával kapcsolatos szabályt úgy foglalhatjuk össze, hogy **if**, valamint **else** után következhet újabb **if**, de **then** után nem.

Befejezésül megemlítünk még két érdekességet a feltételes logikai kifejezésekről:

1. Az első érdekesség, hogy minden feltételes logikai kifejezés átírható feltétlen kifejezéssé. A (3.34) általános alakú feltételes logikai kifejezés ui. a következő feltétlen kifejezéssé írható át:

$$L \wedge K1 \vee \neg L \wedge K2 \tag{3.37}$$

(3.37) alapján pedig bármely feltételes kifejezéslánc visszavezethető feltétlen kifejezésre, de egy ilyen átírás alaposan elbonyolíthatja és áttekinthetetlenné teheti a formulákat. Erről könnyen meggyőződhet az Olvasó, ha megpróbálja pl. a (3.36) feltételes kifejezést átírni.

2. Második megjegyzésünk, hogy a tilalmat, amelyik a relációkból kizárja a feltételes aritmetikai kifejezéseket, a módosított ALGOL-jelentés vezette be. Az eredeti ALGOL-jelentés megengedte, hogy relációk jobb és bal oldalán tetszőleges aritmetikai kifejezések szerepeljenek. Ez lehetővé

tette, hogy valaki szintatikusan korrekt módon, **then** után **if**-et írjon. Pl.:

$$\textbf{if } A \vee B \wedge C \textbf{ then } x + y \textbf{ else } x - y \geqq 0 \qquad (3.38)$$

A reláció a II. fejezet 8. p. szerint *D* típusú részkifejezés, és mint ilyen, korrekt logikai kifejezés. Helyettesítsük (3.38)-at egy (3.34) feltételes logikai kifejezésben *K*1 helyére! E helyettesítés elvégzése után — mint könnyű látni — **then if** kombináció jön létre. A módosított ALGOL-jelentés által bevezetett szintaktikus tilalom kiküszöbölte a **then if**-re vonatkozó ezen egyetlen kivételt is.

A feltételes logikai kifejezések az ALGOL szokványos felhasználási területein ritkán találnak alkalmazásra. Tárgyalásukra inkább csak a teljesség kedvéért tértünk ki; alkalmazásukra igazán gyakorlati példát nehéz lenne találni.

## 5. Összetett utasítások

Az előző pontban említettük, hogy az ALGOL-ban lehetőség van az utasítások zárójelezésére, azaz egy több elemi utasításból álló ALGOL-utasításcsoport — külön e célra szolgáló „zárójelek" segítségével — egyetlen utasítássá fogható össze, hasonlóan az aritmetikai és logikai kifejezésekhez, amelyekben a zárójel egy hosszabb részkifejezést egyetlen egységgé fog össze. Az utasítások zárójelezésére bevezetett alapjeleket utasítás-zárójeleknek nevezzük. A kezdő utasítás-zárójel a **begin** (kezdet), a végzárójel az **end** (vég) alapjel. Bármely ALGOL-utasításcsoport egyetlen feltétlen utasításnak számít, ha utasítás-zárójelek közé zárjuk. A zárójelbe zárt utasítást vagy utasításcsoportot *összetett utasításnak* nevezzük. Az összetett utasításban szereplő egyes utasítások tetszőlegesek lehetnek; így többek között szerepelhet köztük újabb összetett utasítás. Ilyen esetben az utasítás-zárójelek egymásba skatulyázódnak, ami tetszőleges mélységig megengedhető.

Mint a 2. p. egyik lábjegyzetében már jeleztük, az **end** végjel értékű jel, ezért pótolja az előtte álló utasítást bezáró pontosvesszőt. Az **else**-szel ellentétben azonban, **end**

elé szabad fölösleges pontosvesszőt írni. Ez szemantikai szempontból egy üres utasítás beiktatását jelenti.

Az utasítások zárójelezésére az aritmetikai kifejezések zárójelezési szabályaival teljesen analóg szabályok érvényesek. Két fontos szabályt mégis külön kiemelünk:

1. Minden megkezdett zárójelet be kell zárni, azaz minden **begin**-hez tartozik egy és csakis egy **end**.

2. Az *A* és *B* összetett utasítások vagy teljes egészükben idegenek, vagy közülük az egyik teljes egészében tartalmazza a másikat. Tehát megengedettek az alábbi alakok:

$$\underbrace{\textbf{begin}\ \{\text{utasítások}\}\ \textbf{end};}_{A} \qquad \underbrace{\textbf{begin}\ \{\text{utasítások}\}\ \textbf{end}}_{B}$$

és

$$\frac{\textbf{begin}\ \{\text{utasítások}\};\ \underbrace{\textbf{begin}\ \{\text{utasítások}\}\ \textbf{end};}_{B}\ \{\text{utasítások}\}\ \textbf{end}}{A}$$

*Példák*

**1.** Tegyük fel, hogy ki kell számítanunk valamely *P* mennyiséget, amely a következő két képlettel van definiálva:

$$P = \begin{cases} \dfrac{\sqrt[3]{1+x^2-\cos\varphi+x-y}}{\sqrt{1+x^2-y}}, & \text{ha } x>y \\ \dfrac{\sqrt[3]{1+y^2-\sin\varphi+y-x}}{\sqrt{1+y^2-x}}, & \text{ha } x\leqq y \end{cases}$$

Írjunk ALGOL-programot *P* kiszámítására!

Feladatunknak eleget tesz a következő ALGOL-program:

```
if x > y then begin M1 := 1 + x↑2 − y;
        P := sqrt (x + M1 − cos (fi))/M1↑0.3333333 end
else begin M1 := 1 + y↑2 − x;
    P := sqrt(y + M1 − sin (fi))/M1↑0.3333333   end;   {folytatás}
```

Itt *M*1 segédváltozó, amelynek segítségével a képletben szereplő ismétlődő részkifejezések kétszeri kiszámítását kerüljük el. A program egyetlen (3.19.1) típusú feltételes utasítás, amelyben *U*1 és *U*2 helyén összetett utasítások állnak.

2. Készítsünk ALGOL-programot az

$$ax^2+bx+c=0 \tag{3.39}$$

másodfokú egyenlet megoldására! Olyan programot kell készítenünk, amely szétválasztja a valós és a konjugált komplex gyökök esetét, továbbá amennyiben az egyenlet elfajult, de megoldható ($a=0$, de $b\neq 0$), akkor elsőfokú egyenlet módjára oldja meg, és egyidejűleg jelzi az elfajultság tényét (**false** értékűvé tesz egy bizonyos *B* logikai változót). Az $a=b=0$ esetben átadandó a vezérlés egy meghatározatlan *STOP* címkére (feltesszük, hogy ez a címke valahol a program más helyén el van helyezve). A megoldások az *x*1*r*, *x*2*r*, *x*1*i*, *x*2*i* változókba helyezendők el. Feltesszük, hogy a programba való belépéskor *a*, *b* és *c* aktuális értéke meghatározott.

A megoldási algoritmus nagyobb pontossága érdekében valós gyökök esetén a két gyök közül a kisebbik abszolút értékűt nc a megoldóképletből, hanem a gyökök és együtthatók közötti $x_1x_2=c$ összefüggésből számítsuk. Ellenkező esetben ugyanis a kisebbik abszolút értékű gyök indokolatlan jegyveszteséggel adódik. A program:

```
if a=0 ∧ b=0 then go to STOP;                                   (3.40)
if a≠0 then begin M:=b↑2-4×a×c;   P:=sqrt(abs(M));
  if M≧0 then begin x1i:=x2i:=0;   x1r:=(-b+P)/2/a;
    x2r:=(-b-P)/2/a;
    if abs(x1r)>abs(x2r) then x2r:=c/x1r else x1r:=c/x2r
    end else
    begin x1r:=x2r:=-b/2/a;   x1i:=P/2/a;   x2i:=-x1i
    end;
B:=true end else begin x1r:=x2r:=-c/b;
x1i:=x2i:=0;   B:=false end;   {folytatás}
```

A (3.40) program valóságos gyűjteménye a különböző feltételes utasításoknak. Valójában a program mindössze két utasításból áll: az első utasítás egy (3.19.2) típusú feltételes utasítás, amely leválasztja a teljesen értelmetlen $a=b=0$ esetet, a második utasítás pedig maga a program, amelyet egyetlen (3.19.1) típusú feltételes utasítás alkot. $U1$ és $U2$ helyébe itt összetett utasítások kerültek, melyek belsejében ismét vannak feltételes utasítások.

Az összetett utasítások alkalmazására a könyv további fejezeteiben még igen sok példát látunk majd. Amint azt a IV. fejezetben látni fogjuk, az utasítás-zárójeleknek az ALGOL nyelv blokkstruktúrájával kapcsolatban különlegesen fontos szerep jut.

Utasítás-zárójeleket alkalmazhatunk akkor is, ha egy feltételes utasításban más módon nem tudjuk eltüntetni a **then** utáni **if**-et. Az ALGOL szintaxisa megengedi egyetlen ALGOL-utasítás zárójelbe zárását is, de ennek szükségessége csak ebben az egyetlen esetben merül fel.

## 6. Indexes változók

Amikor eddigi tárgyalásainkban változókról beszéltünk, mindig index nélküli, skaláris változókra gondoltunk. Ebben a pontban az indexes változók használatára vonatkozó szabályokat taglaljuk.

A matematikában igen gyakran felmerül a szükségessége annak, hogy bizonyos számcsoportok elemeivel ne csak egyenként tudjunk műveleteket végezni, hanem magukat a csoportokat egészükben is kezelni tudjuk. Ez a helyzet például a lineáris algebra vizsgálatainál, a statisztikában és a matematika sok más ágában is. A számcsoportokon végzett műveletek általában igen nagy menynyiségű aritmetikai és logikai operációt tesznek szükségessé, és ezért elvégzésükre gyakran elektronikus számológépet kell igénybe venni.

Az ilyen feladatokban általában nagy mennyiségű számadaton azonos, vagy igen hasonló műveletek elvégzése

szükséges. (Gondoljunk például a mátrixalgebra műveleteire, adatok rendezésére, átlagok és egyéb statisztikai függvények képzésére stb.)

Eddigi ismereteink alapján ilyen jellegű feladatokat csak igen nehézkesen tudnánk programozni. Ha például azt a feladatot kapnánk, hogy írjunk programot két tízelemű vektor skaláris szorzatának kiszámitására, ezt csak úgy tudnánk megtenni, hogy felírnánk egy hosszú értékadó utasítást, amelyik elemenkénti műveletekre lebontva jelöli ki a kívánt skaláris szorzást:

$$s := a1 \times b1 + a2 \times b2 + a3 \times b3 + a4 \times b4 + a5 \times b5 + a6 \times b6 + a7 \times b7 + a8 \times b8 + a9 \times b9 + a10 \times b10; \qquad (3.41)$$

Ez tízelemű vektor esetén, ha nem is kényelmes, de végszükség esetén elviselhető volna. Százelemű vektor esetén azonban már nagyon nehézkes.

Elvben könnyű látni a kiutat. Olyan programot kell szerkesztenünk, amely ciklusban szorozza össze a vektorok egyes elemeit, és minden egyes ciklusban hozzáad egy-egy szorzatot az előző szorzatok részletösszegéhez. De hogyan tudjuk ezt ALGOL-ban leírni? Ha *a*1, *b*1, *a*2, *b*2... egymástól függetlenül definiált skaláris változók, akkor ezeken nem tudunk végigfuttatni egy indexváltozót, mert az „általános", *i*-edik elemet nem jelölhetjük egyszerűen *ai*-vel.

A felvetett probléma az ALGOL-ban indexes változók alkalmazásával oldható meg. Az indexes változók valamilyen szempontból összetartozó változók által alkotott csoportok elemei. Az összetartozást azzal fejezzük ki, hogy a változócsoport elemeit közös azonositóval látjuk el, és a csoport egyes elemeinek az azonosítón kívül egy vagy több hivatkozási számot is adunk. A hivatkozási számot vagy számokat nevezzük indexeknek. Egy indexes változó akkor van meghatározva, ha azonosítóján kívül valamennyi indexe adott.

Az indexes változókat a skaláris változóktól elkülönítve deklarálják. A deklarációban a változók azonosítóján kívül az egyes indexek értékhatárait is fel kell tüntetni.

Ha például az *F* azonosítóval ellátott változócsoporthoz tartozó egyedi változók két indexet viselnek, és az első index 1-től 12-ig, a második 0-tól 7-ig futhat, akkor az $F_{9,5}$ indexes változóra való hivatkozásnak van értelme, de az $F_{5,8}$-ra való hivatkozás már értelmetlen.

Az indexes változók teljesen úgy viselkednek, mint az index nélküli, skaláris változók. Típusuk, akárcsak a skaláris változóké, lehet egész, valós vagy logikai. Ugyanúgy használhatók kifejezésekben, értékadó utasításokban. Az egyetlen eltérés, hogy az indexes változó aktuális értékéhez való hozzáférés előtt előbb indexeinek aktuális értékeit kell meghatároznunk.

Az indexeket az ALGOL-ban szögletes zárójelbe zárva, egymástól vesszővel elválasztva soroljuk fel. Az index definiálható tetszőleges aritmetikai kifejezés segítségével. Ha az indexet nem számkonstanssal, hanem változóval vagy aritmetikai kifejezéssel adjuk meg, akkor az index számértékét (aktuális értékét) ezeknek a kifejezéseknek, illetve változóknak aktuális értéke határozza meg.

Példák indexes változókra:

$P[i, k]$ $A[i+j-3, (j-s)\lceil 3, i]$

$G[5, 2, j, k, n]$ $B[\textbf{if } a>b \textbf{ then } 2 \textbf{ else } 3]$

Ezekre a változókra való hivatkozásnak nyilván csak akkor van értelme, ha az indexkifejezéseknek van aktuális értékük, és ez az érték az előre definiált értékhatárok közé esik. Magától értetődő követelmény az is, hogy az indexek száma megegyezzék a deklarációkban rögzített indexszámmal. Az indexek száma nincs felülről korlátozva; szabad akárhány indexes változót is deklarálni.

Az előzőekben már említettük az indexes változók deklarálását. Ennek részletes tárgyalására a könyv IV. fejezetében (3. p.) kerül sor. Az indexes változók csoportjait gyűjtőnéven *tömböknek* nevezzük. A tömb speciális esete a vektor (egyméretű tömb, egyindexes változók csoportja), valamint a mátrix (kétméretű tömb, kétindexes változók csoportja).

Néhány további szemantikai szabály az indexes változókkal kapcsolatban:

1. Indexként valós típusú változót vagy valós értékű aritmetikai kifejezést is alkalmazhatunk, bár az index mindig egész szám. Ilyen esetben a valós értékről az egészre az értékadó utasításoknál tárgyalt kerekítési szabály szerint kell áttérni. Az indexet mindig úgy kell tekinteni, mintha ki nem írt egész típusú fiktív változó volna, és az indexkifejezések kiszámítása valójában értékadás erre a fiktív változóra nézve. Pl. az

$$A[i-j+k, (x-y+z)\uparrow 2]$$

indexes változó indexeinek kiszámítása közben két fiktív értékadásra kerül sor, az egyik

$$p1 := i-j+k;$$

míg a másik:

$$p2 := (x-y+z)\uparrow 2;$$

Ha $i=3$; $j=2$; $k=2$; $x=2.2$; $y=0.7$; $z=1.4$, akkor az első index (a $p1$ fiktív változó) aktuális értéke 3, a második index ($p2$) aktuális értéke 8; ugyanis a jobb oldali kifejezés aktuális értéke 8.41, és ezt a „lekerekítési" szabály miatt (l. 2. p.) 8-ra kell lekerekíteni.

2. Külön szemantikai szabály rendelkezik az olyan értékadó utasításokról, amelyek bal oldali változója indexes változó: az ilyen utasításokat oly módon kell végrehajtani, hogy előbb meghatározzuk az összes bal oldali változó indexkifejezéseinek aktuális értékét, mégpedig — ha több indexkifejezés van — szigorúan balról jobbra tartó sorrendben, és ezután a jobb oldali kifejezés aktuális értékét. E szabály nélkül egyes értékadó utasítások értelmezése kétes volna. Például:

$$A[i] := i := i+1; \tag{3.42}$$

Legyen $i$ aktuális értéke 2. A kimondott szabály értelmében (3.42) az $i$, és az $A[2]$ változóknak 3-at ad értékül. A lényeg az, hogy itt az $A[2]$, és nem az $A[3]$ változót kell figyelembe venni. (3.42) ugyanis megváltoztatja az $i$ index-

változó aktuális értékét is, de csak az indexkifejezés aktuális értékének meghatározása után.

3. Egy indexes változó indexkifejezésében tetszés szerinti számú újabb indexes változó szerepelhet. Ezt a szabályt rekurzíve értelmezve, kimondhatjuk, hogy az indexek tetszőleges mélységben egymásba skatulyázhatók.

*Példák*

1. Első példaként vizsgáljuk a fejezet elején felvetett feladatot, két vektor skaláris szorzatának meghatározását. Legyen a két vektor (egyméretű tömb) azonosítója *a* és *b*,* a közös rendszám *n*. Legyen az eredmény az *S* változó értéke. A program:

```
S:=0;  i:=1;
L: if i=n+1 then go to out;  S:=S+a[i]×b[i];          (3.43)
i:=i+1;  go to L;  out:  {folytatás}
```

A jelen pont elején kitűzött feladatot sikerült megoldanunk, az indexes változók segítségével programhurkot generáltunk a skaláris szorzás feladatához. A programot bevezető $S:=0$ utasítás a gyűjtőrekesz törlése. Meg kell szoknunk, hogy ha egy rekeszt (változót) egy szumma részletösszegeinek tárolására használunk, és ennek a rekesznek a tartalmához valamely ciklus minden egyes menetében hozzáadunk egy újabb tagot [amint azt a (3.43) programban az $S:=S+a[i]\times b[i]$ utasítás teszi], ezt a változót a ciklus elején törölnünk kell, azaz nullát kell adnunk értékéül. Ha ez az utasítás nem állna a program elején, az *S* változó úgy kerülne aritmetikai kifejezésbe, hogy előzőleg nem volna definiált értéke, és ezért az egész aritmetikai kifejezés értéke határozatlan volna. Az „*out*" címke a program folytatását jelképezi. A program szerkezete világos; a gép minden egyes ciklusban összeszorozza a vektorok egy-egy elempárját, hozzáadja az eredményt az előző szorzatok

* Tekintettel arra, hogy az ALGOL nyelvben a skaláris változók és a tömbök betűjelölése (azonosítója) írásmód szempontjából semmiben sem különbözik, a továbbiakban másutt sem alkalmazzuk az irodalomban szokásos vastagbetűs írásmódot, sem pedig egyéb megkülönböztetést (kétszeres aláhúzás, föléírt nyíl stb.).

részletösszegéhez, megnöveli *i* értékét, és addig folytatja az eljárást, amíg *i* el nem érte az $n+1$ értéket.

2. Írjunk programot egy *A* négyzetes mátrix és egy *b* vektor szorzatának meghatározására. Az eredmény legyen a *c* vektor. A rendszám *n*. A program:

```
j:=1;   K: if j=n+1 then go to out;
S:=0;   i:=1;   L: if i=n+1 then go to M;
S:=S+A[j, i]×b[i];   i:=i+1;   go to L;
M: c[j]:=S;   j:=j+1;   go to K;   out:   {folytatás}
```
(3.44)

A (3.44) program némi bővítéssel adódik (3.43)-ból. Itt kétszeresen egymásba skatulyázott ciklussal van dolgunk. A belső ciklus rögzített *j* mellett, skalárisan összeszorozza az *A* mátrix *j*-edik sorát a *b* vektorral. A skaláris szorzás programja teljesen megegyezik a (3.43) programmal. E köré a program köré van építve egy külső ciklus, amely egy skaláris szorzás elvégzése után „továbbviszi" eggyel *j* értékét, és a megnövelt *j*-vel megismételteti a skaláris szorzást. Az algoritmus véget ér, ha *j* elérte az $n+1$-et, azaz már az *n*-edik sort is összeszorozta a gép a *b* vektorral.

A példával kapcsolatban még egy megjegyzést kell tennünk. A figyelmes Olvasó bizonyára észrevette, hogy az *S* segédváltozó látszólag felesleges, ha helyébe mindenütt $c[j]$-t írnánk, akkor a $c[j]:=S$ értékadó utasítás fölöslegessé válnék, és egy látszólag rövidebb programhoz jutnánk. Ez meg is felel a valóságnak, és az így kapott program mind szintaktikusan, mind szemantikailag kifogástalan lenne. Publikációs célra készült program esetén ez a rövidítés kívánatos is, mivel fokozza a program áttekinthetőségét. Ha azonban a programot gépre szánjuk, akkor a (3.44) változat időben gazdaságosabb, mert az indexes változókhoz való fordulás mindig hosszabb időt vesz igénybe, mint a skaláris változókhoz való fordulás. Éppen ezért időtakarékosság céljából minden olyan helyen, ahol ezt a program szerkezete megengedi, az indexes változókat skalárisakkal célszerű helyettesíteni.

3. Legyen $b$ egy $n$ elemű vektor, és összetevői legyenek egy polinom együtthatói. (Az $x^{n-1}$ tag együtthatója legyen $b_1$, $x^{n-2}$-é $b_2$ . . ., a konstans tag legyen $b_n$.) Írjunk fel olyan programot, amely a Horner-séma alapján kiszámítja a polinom helyettesítési értékét az $x$ helyen, és az eredményt elhelyezi a $P$ változóban. A Horner-séma szerint, mint ismeretes, a polinomot átrendezzük a következőképpen:

$$(. . .((b_1x+b_2)x+b_3)x. . .)x+b_n.$$

Az átrendezett alak igen kényelmesen programozható, és kevesebb memóriarekeszt igényel, mint ha a polinom helyettesítési értékét a legkézenfekvőbb módon, hatványozásokkal határoznánk meg. A program:

```
S:=b[1];   k:=1;   L: if k=n then go to out;
S:=S×x+b[k+1];   k:=k+1;   go to L;                    (3.45)
out:   {folytatás}
```

A program az $S$ rekeszben gyűjti a szukcesszív szorzások és összeadások részeredményeit; $n$ lépés után $S$ aktuális értéke éppen a polinom helyettesítési értéke. A program tetszőleges $n$-re helyesen működik, még $n=1$ esetén is, amikor a polinom konstanssá fajul el.

Eddigi ismereteink alapján már mindent le tudunk írni ALGOL-ban, ami egyáltalában ALGOL-ban leírható. A továbbiakban a programozással kapcsolatban már csak bizonyos lehetőségeket mutatunk meg a programok kényelmesebb szerkesztésére. Lényegesen újat tartalmaznak viszont a deklarációk megadásának módozataival foglalkozó pontok.

## 7. Ciklusutasítások

Már az eddig tárgyalt példaprogramok között is szerepelnek néhányban különféle programhurkok (ciklusok), többek között a (3.22), (3.43), (3.44), (3.45) programokban. E programokból is látható, hogy a programciklusok három lényeges alkatrészt tartalmaznak, éspedig:

1. A ciklus előkészítése, bizonyos — a ciklusban értéküket megváltoztató — mennyiségek kezdeti értékének beállítása;

2. A ciklusban változó mennyiségek továbbléptetése, értékük megváltoztatása;

3. A ciklusból való kilépés fölött döntő logikai vizsgálat, vezérlésátadás vagy a program folytatására, vagy a ciklusban ismétlendő utasítások további végrehajtására.

A 2. pontba tartozó utasításokat a ciklus „magjának" nevezzük.

A (3.45) program esetében például, a program első két utasítása az előkészítő rész (1. pont), a következő feltételes utasítás alkotja a 3. pontot, a többi utasítás a ciklus magjához tartozik.

Ha összevetjük a (3.43), (3.44), (3.45) programokat, észrevesszük, hogy azok tartalmaznak néhány igen hasonló utasítást. Szerepel bennük ugyanis egy-egy változó, amelyre vonatkozóan valamely logikai feltételnek kell teljesülnie a ciklusból való kilépéshez, és mindhárom programban egyaránt megtalálható e változók kezdőértékének a beállítása, továbbléptetése, és a reá kirótt logikai feltétel teljesülésének ellenőrzése. Ezek az utasítások tulajdonképpen nem vesznek részt az effektív számításban, csupán a program vezérléséhez szükségesek. *Adminisztratív utasításoknak* is szokták őket nevezni. A (3.44) program kissé bonyolultabb a másik kettőnél. Itt két egymásba skatulyázott ciklusunk van, és így a ciklusszervezéssel kapcsolatos adminisztráció is kétszeres.

Annak érdekében, hogy a ciklusok szervezésének adminisztrációs utasításaiba kevesebbet kelljen írni, az ALGOL-ban vannak olyan utasítások, amelyek egyesítik magukban egy kiválasztott változó kezdeti értékének beállítását, ciklusonkénti továbbléptetését, és a ciklusból való kilépéshez szükséges logikai vizsgálatot: az ilyen speciális utasításokat *ciklusutasításoknak* nevezzük. Ebben a pontban a ciklusutasítások különféle típusaival foglalkozunk.

Az ALGOL ciklusutasításainak három alaptípusa van, ezekkel azonban igen sok változat képezhető. Először az

alaptipusokat ismertetjük, majd áttérünk a kombinált típusok tárgyalására.

A ciklusutasítások tárgyalását egy újabb alapjel bevezetésével kezdjük: ez a **for**.* Minden ciklusutasítás ezzel kezdődik.** A cikluskezdet általános alakja a következő:

**for** *V*:= (3.46)

ahol *V* tetszőleges (deklarált) változó, és a := jel a már ismert értékadási jel. A (3.46) cikluskezdet utáni jelek a ciklusutasítás típusa szerint különböznek.

### *a)* A ciklusutasítás első alaptípusa

A ciklusutasítás első alaptípusában a cikluskezdetet a következő konstrukció követi:

*A*1 **step** *A*2 **until** *A*3 **do** *U*; (3.47)

ahol *A*1, *A*2 és *A*3 tetszőleges aritmetikai kifejezések, a **step**,*** az **until****** és a **do*******  pedig újabb ALGOL-alapjelek. A **do** elhatároló jelet valamilyen *U* utasítás követi. *U* általában összetett utasítás, és a ciklus magját képviseli.

Áttérünk a (3.47) utasítás szemantikájának tárgyalására.

Az utasítás végrehajtása azzal kezdődik, hogy a *V* változó, amelyet a továbbiakban *ciklusváltozónak* fogunk nevezni, felveszi az *A*1 aritmetikai kifejezés aktuális értékét. Ezt követőleg a gép megvizsgálja, hogy *V* értéke elérte-e már *A*3 értékét. (Később pontosan definiálni fogjuk, hogy mit értünk ezen.) Ha *V* még nem érte el *A*3 értékét, akkor végrehajtódik az *U* utasítás. *U* végrehajtása után *V* értéke megnövelődik *A*2 aktuális értékével, és a gép ismét megvizsgálja, hogy elérte-e már *A*3-at. Ha nem,

* **for** = részére, számára

** Emiatt a ciklusutasítást magyarul zsargonban, de az angol terminussal megegyezően, **for** utasításnak is szokták nevezni.

*** **step** = lépés, lépj.

**** **until** = -ig.

***** **do** = tedd, hajtsd végre.

akkor ismét végrehajtódik $U$, ismét megnövelődik $V$ értéke, és ismét vizsgálat következik, hogy elérte-e már $V$ az $A3$-at. A ciklus addig ismétlődik, amíg csak $V$ el nem éri $A3$-at; ezután a gép áttér a ciklusutasítás utáni utasítás végrehajtására.

Az $U$ utasítás általában (de nem feltétlenül) valamilyen formában függ a $V$ ciklusváltozótól.

*Példák*

**1.** Első példaként tekintsük ismét a skaláris szorzás programját. Tanulságos összehasonlítani ezt a programváltozatot (3.43)-mal, hogy lássuk, mennyiben egyszerűsíti le a programot a ciklusutasítások bevezetése. A program:

$$S:=0; \tag{3.48}$$

**for** $i:=1$ **step** 1 **until** $n$ **do** $S:=S+a[i]\times b[i]$; {folytatás}

A program szerkezete ugyanaz, mint (3.43) esetében, azonban az $i$ változó módosításáról, kezdőértékének beállításáról és a kilépési vizsgálatról „egyszemélyben" a **for** utasítás gondoskodik.

Megjegyezzük, hogy a **step** ... **until** típusú ciklusban az $U$ utasítás még akkor is végrehajtódik, amikor a ciklusváltozó értéke éppen egyenlő az $A3$ kifejezés aktuális értékével (példánkban $i=n$-nel).

Egymásba skatulyázott ciklusok oly módon képezhetők, hogy az $U$ utasítás valamelyik utasítása helyébe újabb **for** utasítást írunk. Írjuk át például **for** utasítás segítségével a (3.44) programot!

**2.** Négyzetes mátrix és vektor szorzatának számítása. (3.44)-nek ciklusutasítások segítségével való átírása után a következő adódik:

**for** $j:=1$ **step** 1 **until** $n$ **do begin** $S:=0$;
**for** $i:=1$ **step** 1 **until** $n$ **do** $S:=S+A[j, i]\times b[i]$; (3.49)
$c[j]:=S$ **end**; {folytatás}

A (3.49) program két egymásba skatulyázott ciklusból áll. A külső ciklus a $j$ sorindex szerint halad, és magja egy

összetett utasítás. A ciklus magján belül egy újabb, *i* szerinti ciklusutasítást találunk. Ez utóbbinak magja egyetlen értékadó utasításból áll, ezért a második **do** után nem következik **begin**. Jól jegyezzük meg: a ciklusutasítás magja lehet elemi vagy összetett utasítás. Ha a ciklusmag egyetlen elemi utasításból áll, akkor a **do** után nem áll **begin**. Ez a helyzet a (3.49) program *i* szerinti ciklusánál. Ezzel szemben, ha a ciklus magja összetett utasítás, akkor a **do** után **begin** következik. (Ha speciálisan a ciklusutasítás magja egyetlen újabb ciklusutasítás, akkor nem kell a **do** után **begin**, mert a ciklusutasítás, magjával együtt, elemi utasítás, még akkor is, ha a ciklusmag összetett utasítás.)

**3.** Készítsünk programot, amely az

$$f(x) = \frac{1 + 2x^2}{\sqrt{1 + x + \sin^2 x}}$$

függvény értékeit összegezi 0-tól 1-ig, 0.01 lépésközzel. A program:

```
S:=0;
for x:=0 step 0.01 until 1 do
    S:=S+(1+2×x↑2)/sqrt(1+x+sin(x)↑2);   {folytatás}
```

Programunkban a ciklusváltozó valós típusú és a lépésköz sem egész szám, ellentétben az előző két példával. Az ALGOL megengedi akár egész, akár valós típusú ciklusváltozó alkalmazását.

Visszatérünk most az első alaptípusú ciklusutasítás szemantikájának tárgyalására.

Bizonyos ALGOL-jelsorozatok szemantikájának pontos definíciója igen sok szabály leírását teszi szükségessé. Ezért helyenként célszerű a szemantikai leírás egy másik módját választani. Megtehetjük azt, hogy valamely nem ismert szemantikájú ALGOL-programot valamely már ismert szemantikájú ALGOL-program segítségével definiálunk. Ezt az utat követjük a ciklusutasítások esetében. Definíciószerűen a (3.47) ciklusutasítás hatása egyenértékű a következő ALGOL-program hatásával:

```
V:= A1;
L:if (V− A3)× sign (A2)>0 then go to out;
U utasítás;                                        (3.50)
V:= V+ A2;  go to L;
out:  {folytatás}
```

A *sign* (*A*2) szorzótényező biztosítja a kilépési feltétel pontos megfogalmazását abban az esetben is, amikor a lépésköz negatív, tehát a ciklusváltozó felülről lefelé közelíti meg az *A*3 értéket.

A (3.50) program igen hasznos abban az esetben, ha a programozónak kételyei merülnek fel valamely elsőfajú ciklusutasítás pontos jelentését illetően. Ebben az esetben csak meg kell vizsgálnia, mit hajt végre a program, ha a (3.50) definíciós program szerint ciklusutasítást nem tartalmazó programmá írjuk át. Például a programozó azt szeretné tudni, mit csinál az elsőfajú ciklusutasítás, ha a ciklusváltozó soha nem veszi fel pontosan az *A*3 értéket, hanem „átlépi" azt. A (3.50) program választ ad erre a kérdésre: a ciklus magja (*U*) azzal az értékkel hajtódik végre utoljára, amelyet az átlépést közvetlenül megelőzően viselt a ciklusváltozó. Az átlépés utáni értékkel *U* már nem hajtódik végre.

Az általunk felhozott példák eléggé speciálisak abból a szempontból, hogy azokban az *A*1, *A*2, *A*3 aritmetikai kifejezések mind számkonstansok vagy változók. Általános esetben ezek tetszőleges aritmetikai kifejezések lehetnek, amelyek értéküket változtathatják a ciklus minden egyes lefutásakor, hiszen az *U* utasítás értéket adhat bármelyik *A*1-et, *A*2-t vagy *A*3-at felépítő változónak. Ilyen módon keletkező bonyolultabb programok szemantikájának tisztázására is (3.50)-hez kell fordulni.

### *b)* A ciklusutasítás második alaptípusa

A **for** utasítás második alaptípusában a ciklusfej jelei után a következő konstrukció áll:

*A* **while** *B*  (3.51)

ahol $A$ aritmetikai kifejezés, $B$ logikai kifejezés és a **while*** szó új alapjel. A (3.51) jelsorozattal képezett, másodfajú ciklusutasítás a következő alakú:

$$\textbf{for } V := A \textbf{ while } B \textbf{ do } U \tag{3.52}$$

ahol $U$ tetszőleges utasítás.

Szemantikai szempontból a (3.52) utasítás jelentése a következő:

1. A $V$ változónak átadandó az $A$ aritmetikai kifejezés aktuális értéke.

2. Megvizsgálandó a $B$ logikai kifejezés aktuális értéke.

3. Ha $B$ aktuális értéke **true**, akkor végrehajtandó az $U$ utasítás, és ezután átadandó a vezérlés a $V := A$ értékadásra, azaz megismétlendő a $V$ változóra vonatkozó értékadás és a logikai vizsgálat.

4. Ha $B$ aktuális értéke **false**, akkor át kell térni a soron következő utasítás végrehajtására.

Végeredményében a (3.52) ciklusutasítás azt jelenti, hogy mindaddig ismételni kell az $U$ utasítás végrehajtását, amíg csak a $B$ kifejezés **true** értékű marad, és amint ez az érték **false**-ra változik, át kell térni a következő utasításra.

A (3.52) ciklusutasítás szemantikáját az alábbi ALGOL-program segítségével definiáljuk:

```
L:V:= A;
if ¬ B then go to out;
U utasítás; go to L;                    (3.53)
out: {folytatás}
```

A (3.52) ciklusutasítással kapcsolatos mindenfajta szemantikai kérdésben a (3.53) program az irányadó, hasonlóan a (3.50) programhoz, amelynek segítségével a (3.47) ciklusutasítás szemantikáját definiáltuk.

A (3.52) típusú ciklusutasítás jól használható abban az esetben, ha nem tudjuk előre, hányszor kell a ciklust

* **while** = mindaddig, amíg.

lefuttatni. Ez a helyzet például iterációs eljárások programozása esetén, amikor a ciklusok száma az iteráció konvergenciájának gyorsaságától függ.

*Példák*

**1.** A négyzetgyökvonás programját egy változatban már elkészítettük, a feltételes utasítások alkalmazásának illusztrálására (l. 4. p.). Most bemutatjuk a program egy újabb változatát, amelyben a ciklus szervezését **for** utasítás látja el:

```
xe:=1;                                                    (3.54)
for x:=0.5*(xe+A/xe) while abs (x-xe)≧10-4 do
    xe:=x;   {folytatás}
```

A program logikai szerkezete ugyanaz, mint első változaté. Az első utasítás beállítja az iteráció kezdőértékét. Az iterációs ciklus szervezését teljes egészében a ciklusutasítás végzi, ezért a program rövidebb lett: eltűnt a külön logikai vizsgálat és a ciklus elejére történő vezérlésátadás.

**2.** Második példaként tekintsük egy sorfejtés számításának programját. Mint ismeretes, az $e^x$ exponenciális függvény az alábbi végtelen sorba fejthető:

$$e^x = 1+\frac{x}{1!}+\frac{x^2}{2!}+\ldots+\frac{x^n}{n!}+\ldots \qquad (3.55)$$

A (3.55) sorfejtés tetszőleges $x$-re konvergens, azonban minél nagyobb $x$, annál több tagig kell elmenni a sorfejtésben adott pontosság eléréséhez. Feladatunk olyan program írása, amely biztosítja, hogy a gép annyi tagot vegyen figyelembe a sorfejtésből, amennyi egy megadott — pl. $10^{-4}$ — abszolút pontosság eléréséhez szükséges. A program:

```
i:=0;  S:=0;  k:=1;
for i:=i+1 while k≧10-4 do begin                          (3.56)
    S:=S+k;  k:=k*x/i  end;  {folytatás}
```

A (3.56) program nem igényel sok magyarázatot; $S$ a sorfejtés aktuális részletösszege, $k$ pedig az újabb tag, amelyet

a gép mindaddig hozzáad a megelőző tagok részletösszegéhez, amig $k$ elegendően kicsinnyé nem válik. Szükségesnek tartjuk még felhívni a figyelmet két apróságra:

1. Emlékeztetünk arra, hogy értékadó utasítások bal oldalán nem szerepelhet egyidejűleg valós és egész típusú változó; tehát ha $S$-ről azt tesszük fel, hogy valós típusú, $i$-ről pedig azt, hogy egész, akkor nem írhatjuk őket egyetlen értékadó utasítás bal oldalára. Ha viszont $i$ is valós típusú, akkor a program első két utasítása összevonható.

2. A programban hallgatólagosan feltettük, hogy $x$ pozitív. (Hol van ez kihasználva? Miként lehetne ettől a feltételtől minél egyszerűbben megszabadulni?)

Hangsúlyozzuk, hogy mindkét példa csak illusztratív jellegű, hiszen mind a négyzetgyökvonást, mind az exponenciális függvény számítását az ALGOL-ban standard függvények végzik, ezért ezeknek ALGOL-ban való programozására soha nincs szükség. (Kivéve, ha valaki éppen ezeknek a számítására akar programot publikálni.)

### *c)* A ciklusutasítás harmadik alaptípusa

A **for** utasítás harmadik alaptípusában a ciklusfejet alapjeleit aritmetikai kifejezések egy véges sorozata követi; az egyes aritmetikai kifejezéseket vesszők választja el egymástól. A harmadik típusú ciklusutasítás alakja a következő:

$$\textbf{for } V := A1, A2, \ldots, An \textbf{ do } U; \qquad (3.57)$$

ahol $U$ utasítás, $A1$, $A2$, ... $An$ pedig tetszőleges aritmetikai kifejezések. A három pont természetesen nem ALGOL-alapjel, ezért (3.57) nem korrekt ALGOL-kifejezés, csupán szimbolizálja a harmadik alaptípusba tartozó ciklusutasítás alakját.

*Példa*

**for** $y := p + 2 \times q$, $x \uparrow 2 + y \uparrow 2$, **if** $x > 1$ **then** $x$ **else** $1/x$ **do** $U$;

ahol $U$ valamilyen ALGOL-utasítás.

(3.57) szemantikáját a következőképpen definiáljuk: a *V* ciklusváltozó felveszi az *A*1 aritmetikai kifejezés aktuális értékét, és ezután végrehajtódik az *U* utasítás. Ezt követőleg *V* felveszi *A*2 aktuális értékét, és ismét végrehajtódik *U*. Általában a (3.57) ciklusutasitás végrehajtásánál a gép sorraveszi az *A*1, *A*2, ... *An* kifejezéseket, rendre átadja ezek értékét a *V* ciklusváltozónak, és minden egyes értékkel egyszer végrehajtja az *U* utasítást. Végeredményben az *U* utasítás annyiszor hajtódik végre, ahány elemet az aritmetikai kifejezések listája, az ún. cikluslista* tartalmaz. A **do** alapjel szerepét úgy definiálhatjuk, mint elhatároló jelet, amely a cikluslista végét jelzi.

A (3.57) ciklusutasitást abban az esetben tudjuk jól alkalmazni, ha nem adható meg kényelmesen olyan képletszerű összefüggés, amely szerint a ciklusváltozó változik, hanem annak valamely önkényesen választott értékrendszeren kell végigfutnia.

*Példa*

Tegyük fel, hogy tabelláznunk kell az

$$f(x) = \frac{1}{\sqrt{1 + e^{-x^2} \cos^2 x}} \tag{3.58}$$

függvényt a következő argumentumértékekre:

$x = 0.05, 0.07, 0.12, 0.2, 0.25, 0.3, 0.5, 1.0, 2.0, 5.0.$

Ennek a feladatnak a megoldására a következő ALGOL-program készithető:

```
i:=1;                                                          (3.59)
for x:=0.05, 0.07, 0.12, 0.2, 0.25, 0.3, 0.5, 1.0, 2.0, 5.0 do
   begin f[i]:=1/sqrt (1+exp(-x↑2)×cos(x)↑2);
   i:=i+1 end;   {folytatás}
```

A (3.59) programban ismét jól láthatjuk az ALGOL nyelvnek azt az előnyét, hogy a programok megírásához alig

* Feltéve, hogy valamely vezérlésátadás előbb ki nem vezet a ciklus magjából.

kell valamit változtatni a feladat megfogalmazása során használt jelöléseken. Ez fokozza a programok áttekinthetőségét, és megkönnyíti a programozási hibák felderítését.

### *d)* Az általános típusú ciklusutasítás

Eddig speciális ciklusutasításokkal ismerkedtünk meg. Most a lehető legáltalánosabb típusú ciklusutasításokkal foglalkozunk.

Az általános **for** utasítás a következő alakú:

**for** $V:=E1, E2, \ldots En$ **do** $U$; (3.60)

ahol $U$ tetszőleges ALGOL-utasítás, $E1$, $E2$, ... $En$ pedig ún. *ciklusLista-elemek.*

Utóbbiak háromfélék lehetnek:

1. $A1$ **step** $A2$ **until** $A3$ alakú, ún. „**step** ... **until**" elemek;
2. $A$ **while** $B$ alakú, ún. „**while**" elemek;
3. Aritmetikai kifejezések.

*Példa*

**for** $i:=1$ **step** 1 **until** $N$, $2\times i$ **while** $i\uparrow 2\leqq 5\times N$ **do** $U$; (3.61)

Itt $U$ valamilyen — számunkra e pillanatban érdektelen — utasítás. A (3.61) utasítás ciklusListája két elemből áll, melyek közül az egyik „**step**...**until**", a másik „**while**" elem. A **for** utasítások listája tetszőleges számú elemet tartalmazhat.

Szemantikai szempontból (3.60) jelentése a következő. Az utasítás végrehajtásakor a gép sorra „feldolgozza" a ciklusLista elemeit, azaz minden egyes ciklusLista-elemnek megfelelően egyszer vagy többször (de esetleg egyszer sem) végrehajtja a ciklus magját képező $U$ utasítást. Ha a soronlevő ciklusLista-elem

1. **step**...**until** elem, akkor az utasítás végrehajtását a (3.50) definíciós program szerint végzi, azaz addig növeli (csökkenti) a ciklusváltozót az $A2$-nek megfelelő lépéssel, amíg az $A3$ aktuális értékét el nem érte;

2. **while** elem, akkor a cikluslista-elem feldolgozása során a (3.53) program szerint jár el, azaz mindaddig ismétli a ciklus magját, amíg a cikluslista-elemben szereplő *B* logikai kifejezés **true** aktuális értékű;

3. aritmetikai kifejezés, akkor átadja a ciklusváltozónak az aritmetikai kifejezés aktuális értékét, és végrehajtja egyszer az *U* utasítást.

A **for** utasítás végrehajtása akkor fejeződött be, ha a cikluslista-elemek feldolgozása során a gép elérkezett a **do**-ig. A **do** elhatároló jel szerepe itt is a cikluslista lezárása, amint azt a (3.57) speciális esetnél is láttuk.

A (3.47), (3.52) és (3.57) ciklusutasítások, mint mondottuk, (3.60) speciális esetei.

*Példák*

1. Tegyük fel, hogy tabellázznunk kell az

$$f(t) = A_1 + A_2 e^{-A_3 t}$$

függvényt, mégpedig a $0 \leqq t \leqq 1$ intervallumban tizedenként, az $1 \leqq t \leqq 10$ intervallumban egészenként, ezenkívül a 20, 50 és 100 helyeken. A **for** utasítások általunk eddig ismert alaptípusai segítségével csak nehézkesen tudnánk a feladatot megoldani. A (3.60) általános ciklusutasítás viszont igen hatékony eszközt nyújt a feladat megoldására:

```
i:=1;
for t:=0 step 0.1 until 1, 2 step 1 until 10, 20, 50, 100 do
   begin f[i]:=A1 + A2 × exp (−A3 × t);   i:=i+1 end;
   {folytatás}                                              (3.62)
```

A cikluslista első két eleme itt **step** ... **until** típusú; a többi számkonstans, azaz aritmetikai kifejezés. Az első két elem biztosítja a függvény tabellázását a két ekvidisztáns számközben, a három számkonstans pedig a függvényértéknek a fennmaradó értékek melletti kiszámítását írja elő.

A (3.62)-höz hasonló programok megírásakor ügyelnünk kell arra, nehogy a ciklusváltozó bizonyos értékeire kétszer futtassuk le a ciklust. Tipikusan ilyen hiba lenne

például a cikluslista második elemében $A$1-ként 2 helyett 1-et írni. Ebben az esetben ui. a ciklus magja $t=1$ mellett kétszer kerülne lefuttatásra.

2. A (3.56) példában az exponenciális függvény sorfejtésének számítását programoztuk. Láttuk, hogy a második alaptípus egymagában nem tudja beállítani ciklusváltozója kezdőértékét. Ugyanis a (3.53) programban az $A$ aritmetikai kifejezés minden gyakorlati esetben közvetlenül vagy közvetve függ $V$-től. (Javasoljuk az Olvasónak: gondolja meg, hogy ellenkező esetben az egész **for** ciklus értelmetlenné válik.) Ezért $A$ értékének a ciklusba való belépéskor definiálva kell lennie, ami megköveteli $V$ értékének előzetes beállítását. Egységesebb és áttekinthetőbb a ciklus szervezése, ha ezt is a **for** utasítás végzi, és nem kell erre a célra egy külön értékadó utasítást alkalmaznunk. Igen sok esetben erre lehetőség nyílik az általános típusú **for** utasítás alkalmazása esetén. Az exponenciális függvény programja például a következőképpen módosítható:

```
S:=0;  k:=1;
for i:=1, i+1 while k ≧10 −4 do                    (3.63)
   begin S:=S+k;  k:=k × x/i end;  {folytatás}
```

A (3.63) program ugyanazt hajtatja végre a géppel, mint a (3.56) változat, azonban az utasítások száma eggyel kevesebb.

(3.63)-hoz hasonló ciklusutasítást igen gyakran alkalmazunk a gyakorlati programozásban, ezért talán hasznos, ha szemantikáját külön is tárgyaljuk, annak ellenére, hogy csak speciális esete (3.60)-nak: a gép először mindenféle logikai vizsgálat mellőzésével végrehajtja a ciklus magját $i=1$ mellett, majd megnöveli $i$ értékét eggyel, és ezután kezdi vizsgálni a $k \geqq_{10} -4$ logikai feltétel teljesülését. Ha tehát valamilyen oknál fogva a ciklus magját egyszer sem szabad végrehajtani, akkor (3.63) típusú utasítás nem alkalmazható.

Van a ciklusutasítások alkalmazásával kapcsolatban néhány olyan szemantikai szabály, amelyeket eddig még nem közöltünk. Most áttérünk ezek ismertetésére:

1. Az első ilyen szabály a *ciklusváltozó végértékéről* rendelkezik. Eszerint:

*a)* Ha a program vezérlése a ciklusból oly módon lép ki, hogy a cikluslista teljesen kimerült, akkor a ciklusváltozó végértéke nem őrződik meg. Más szóval, ha a gép egyszer végrehajtott egy ciklusutasítást, akkor a ciklusból való kilépés után a ciklusváltozó aktuális értékére való mindennemű hivatkozás értelmetlen. Ezt tréfásan úgy szokás megfogalmazni, hogy a ciklusváltozóra nézve a **for** utasítás „értékvesztő utasítás", mert a ciklusváltozót megfosztja értékétől.

*b)* Ha ezzel szemben a vezérlés a ciklusból nem a ciklus kimerülése miatt lép ki, hanem valami más módon, akkor a ciklusváltozónak megőrződik az utolsó értéke, és a továbbiakban is felhasználható. Ezt a körülményt igen jól ki tudjuk használni, a ciklusokból való „vészkijáratok" programozásában. („Egyéb módon" való kilépésen általában **go to** utasítással való kilépést kell érteni.)

Felhívjuk a figyelmet arra, hogy ez a szabály ellentmond a (3.50) és (3.53) ALGOL-programok szemantikájának, mert ezekben a programokban a ciklusból való normális kilépés után is marad értéke a ciklusváltozónak. Amikor azt mondjuk, hogy az első, illetőleg második alaptípusú ciklusutasítást a (3.50) és (3.53) programok segítségével definiáljuk, akkor mindig hozzá kell értenünk ehhez a kijelentéshez az előbb említett kivételt.

2. A következő szabály a *ciklusutasításokba való vezérlésátadásokkal* kapcsolatos. Ezen szabály szerint *egy* **for** *utasítás magját képező U utasításra vagy annak belsejébe mindenfajta vezérlésátadás tilos.* Ezt a szabályt úgy is megfogalmazhatjuk, hogy ha a ciklus magját képező *U* utasítás összetett utasítás, és akár maga az *U* utasítás, akár valamely belső utasítása, címkét visel, akkor ezekre a címkékre csakis olyan vezérlésátadó utasítás hivatkozhat, amely maga is része az *U* utasításnak. Ezt a tilalmat azzal magyarázhatjuk meg, hogy a ciklus belsejébe való „beugrás" következtében a gép átugorná a cikluskezdetet, amelyben a számára lényeges információ el van helyezve.

Az előző szemantikai szabály második részét, és a jelen szabályt összevetve kimondhatjuk, hogy ciklusból **go to** utasítással kiugrani szabad, de abba beugrani nem. Szükségesnek tartjuk ez utóbbira külön nyomatékosan felhívni a figyelmet, ugyanis gépi és autokódokban általában nincs ilyen korlátozás, és a programozók gyakran alkalmazzák azt a fogást, hogy programhiba javításakor valahová elugratnak, ott végrehajtatnak egy kis javítóprogramot, majd visszatérnek az elugrás helyét követő utasításra.*

3. Az alább ismertetendő szabály a *feltételes utasítások és a ciklusutasítások kombinálásával* kapcsolatos. Ez azt mondja ki, hogy a (3.19.1) típusú feltételes utasításban, az *U*1 utasítás helyébe nem szabad **for** utasítást írni. A (3.19.2) feltételes utasítással kapcsolatban nincs ilyen korlátozás. Ezt a tilalmat úgy fogalmazhatjuk meg, hogy a **then** utáni **for** csak abban az esetben van megengedve, ha ugyanahhoz a **then**-hez nem tartozik még egy **else** is. Az **else** és a **for** alapjelek ebben az értelemben kizárják egymást: *ugyanahhoz a* **then***-hez nem tartozhat egyszerre* **else** *is és* **for** *is.*

Ezt a tilalmat a módosított ALGOL-jelentésben vezették be, mert kiderült, hogy ha (3.19.1)-ben *U*1 **for** utasítás is lehet, akkor kétértelmű ALGOL-programok felírására nyílik lehetőség.

Példa erre a következő ALGOL-utasítás:

**if** $a \geqq b$ **then for** $i := 1$ **step** 1 **until** $N$ **do** (3.64)
  **if** $c \geqq d$ **then** $p[i] := 0$ **else** $q[k] := 0$;

A (3.64) programban a kétértelműség nagyon hasonló jellegű a (3.25) példán bemutatotthoz. Itt sincs egyértelműen meghatározva, hogy a két **then** közül melyikhez tartozik az **else**. (3.64)-et ugyanis értelmezhetjük akár így:

**if** $a \geqq b$ **then begin for** $i := 1$ **step** 1 **until** $N$ **do** (3.65.1)
  **if** $c \geqq d$ **then** $p[i] := 0$ **else** $q[k] := 0$ **end**;

akár így:

* Bizonyos korlátozások mellett ezt pl. a FORTRAN nyelv is megengedi, az ALGOL-ban azonban tiltva van.

**if** $a \geqq b$ **then begin for** $i := 1$ **step** 1 **until** $N$ **do** (3.65.2)
**if** $c \geqq d$ **then** $p[i] := 0$ **end else** $q[k] := 0$;

Az új ALGOL-jelentés tisztázta a (3.64) típusú programok szemantikáját: mégpedig az egyetlen lehetséges értelmezés (3.65.1), mivel az a körülmény, hogy a **then** után **for** következik, egyértelműen kizárja annak lehetőségét, hogy az **else** ugyanehhez a **then**-hez tartozzék. Utasítás-zárójelek alkalmazásával természetesen a (3.65.2) értelmezést is el tudjuk érni.

4. Következő megjegyzésünk nem annyira a **for**, mint inkább a feltételes utasítások szemantikájára vonatkozik. A (3.19.1) és (3.19.2) típusú utasítások szemantikai magyarázatában több helyen szerepelt az a félmondat, hogy „... utána pedig át kell térni a feltételes utasítást a felírás sorrendjében követő utasítás végrehajtására." Jogos a kérdés: mi történik akkor, ha a feltételes utasítás egy ciklusutasítás magjának utolsó utasítása, mire kell ebben az esetben áttérni? Tekintsük például a következő programot:

**for** $i := 1$ **step** 1 **until** $N$ **do if** $a[i] \geqq b[i]$ **then**
**begin** $m := b[i]$; $b[i] := a[i]$; $a[i] := m$ **end**; (3.66)
$L$: $x := 2 \times y$; {folytatás}

A probléma a következő. Tegyük fel, hogy már $i = 1$-re nem teljesül az $a[i] \geqq b[i]$ egyenlőtlenség. Ha szó szerint vesszük azt a szabályt, hogy a logikai feltétel nem teljesülése esetén a felírás sorrendjében következő utasításra kell áttérnünk, akkor ez azt jelenti, hogy az $L$ címkéjű utasítást kell végrehajtanunk, amely már kivül esik a ciklus magján. Vagyis kikerültünk a ciklusból anélkül, hogy lejárt volna a ciklus. Kérdés, hogy jogos-e ez a kilépés?

Nem, mert mint a (3.50) és (3.53) programoknál láttuk, a ciklus magját mindig egy hallgatólagos **go to** utasítás zárja le, amely nincsen kiírva, de amelyet beleértünk a ciklus szervezésébe. (Általános ciklusutasítással kapcsolatban ugyan nem írtunk fel definíciós programot, de a helyzet ott is hasonló.) Végeredményben a (3.66) programban

is odaértjük a ciklus magja után a **go to** utasítást, tehát az **if** utasítás utáni utasítás nem az *L* címkéjű utasítás, hanem a ki nem írt **go to**, amely visszaugrat a ciklus magjának elejére, és nem engedélyez rendkívüli kilépést a ciklusból.

A **for** utasítással kapcsolatos összes eddigi tárgyalásainkban hallgatólagosan feltettük, hogy a ciklusváltozó nem indexes változó. Az ALGOL-ban semmiféle szintaktikus szabály nem írja elő, hogy ciklusváltozóként csak index nélküli változók alkalmazhatók. Indexes változók ciklusváltozókénti alkalmazásával teljesen szokatlan programozási fogásokra nyílik lehetőség. Tekintsük például a következő ALGOL-programot:

•

```
i:=1;
for a[i]:=a[1]+b[1] step b[i] until a[n] do                    (3.67)
    i:=i+1;   {folytatás}
```

Egyszerűség kedvéért feltesszük, hogy az $a_i+b_l$ összegek mind kisebbek, mint $a_n$, a $b$ vektor elemei pedig mind pozitívak. E feltevések mellett a (3.67) program — kissé szokatlan módon — az $a$ és $b$ vektorok összegét számítja ki, és az eredményt elhelyezi az $a$ vektorban. Erről könnyen meggyőződhetünk, ha a (3.67) programra alkalmazzuk a (3.50) definíciós programot:

```
i:=1;   a[i]:=a[1]+b[1];
L: if (a[i]-a[n])×sign (b[i])>0 then go to out;                (3.68)
i:=i+1;   a[i]:=a[i]+b[i];   go to L;   out:   {folytatás}
```

A (3.68) program működésének végiggondolása közben azt vesszük észre, hogy indexes változónak ciklusváltozóként való alkalmazása lehetőséget ad a ciklusváltozó menetközbeni megváltoztatására. Amikor a ciklusváltozó index nélküli, akkor a növekményt [amit (3.50)-ben *A*2-vel jelöltünk] mindig ugyanahhoz a változóhoz adtuk hozzá; (3.68) esetében viszont a ciklus minden egyes manetében egy másik változót növeltünk meg. Ezt azáltal értük el, hogy a ciklusváltozó indexkifejezését megváltoztattuk; a ciklus magja értéket ad az *i* változónak, amely a ciklus-

változó indexében szerepel. Éppen ez a (3.68) példa lényege: a ciklus végén már egy egészen más változó a ciklusváltozó, mint amelyik a ciklus kezdetén volt.

A (3.68)-hoz hasonló fogásokkal ügyes programokat lehetne írni. Sajnos azonban az ilyenfajta ciklusutasítások szemantikája nincs végérvényesen tisztázva. Emlékeztetünk például arra, hogy a ciklusváltozó végértéke nincs definiálva a ciklusból való normális kilépés után. Éppen ezért, az ALGOL nem rendelkezik egyértelműen arról, hogy a (3.68) program megőrzi-e valóban az *a* vektorban az $a+b$ vektorösszeget, vagy pedig kiszámítás után az eredmény mindjárt el is vész; általában nincsen tisztázva ui., hogy ha nem egy ciklusváltozó van, hanem több, akkor a végértékről mondottak mindegyikre vonatkoznak-e?

Egyes fordítóprogramok oly módon vágják ketté a gordiusi csomót, hogy kimondják: a ciklusváltozó lehet ugyan indexes, de az indexkifejezés aktuális értéke csak egyszer kerül kiszámításra a ciklus elején, nem pedig a ciklus minden egyes lefutásakor. Ezzel valóban megoldódnak a szemantikai problémák, de mivel elvész egy sereg programozási lehetőség, nem látszik rokonszenves megoldásnak. A módosított ALGOL-jelentésben utalás történik arra, hogy ebben az ez idő szerint nyitott kérdésben idővel dönteni fognak.

## 8. Programozási példák

Az eddigiekben megismertük az ALGOL nyelv csaknem valamennyi programozási lehetőségét. Ebben a pontban néhány programozási feladatot oldunk meg.

A gyakorlatban felmerülő programozási feladatokat igen durván — a következő csoportokba sorolhatjuk:

1. *Adatrendezési feladatok*. Ide tartoznak azok a feladatok, amelyekben egy adatsokaság elemeivel kell különféle — főként logikai jellegű — műveleteket végeznünk. (Adathalmazból adott tulajdonságú elem kiválasztása, adathalmaznak adott feltétel, pl. nagyság szerinti rendezése stb.)

2. *Lineáris algebrai feladatok.* Ide tartoznak a mátrix- és vektoralgebrai feladatok, továbbá a statisztikai adatfeldolgozás különböző feladatai; például átlagok, szórások számítása, statisztikai próbák programjai.

3. *Egyéb numerikus alkalmazások:* transzcendens egyenletek gépi megoldása, differenciálegyenletek megoldása, integrálás, függvényközelítés stb.

4. *Nem numerikus alkalmazások:* gépi fordítás mesterséges és természetes nyelvekről, mesterséges intelligencia gépi programozása,* szabályozási és vezérlési feladatok** stb.

A fenti felosztás igen hozzávetőleges, hiszen egy-egy bonyolultabb gyakorlati feladat egyes részei különböző jellegűek, és nem lehet őket teljes egészükben egyetlen fenti osztályba sem besorolni. Másrészt az egyes kategóriák között végtelen sok különféle átmenet képzelhető el, és maguk az egyes kategóriák sem válnak szét élesen; többek között az első és a második kategória is igen közel fekszik egymáshoz.

Az adott osztályozás a programokra vonatkozik és nem a matematikai modell jellegére. A statisztikai adatfeldolgozás és a lineáris algebra például igen hasonló jellegű feladatokat ad a programozónak, bár ezek matematikailag nem azonosak. A többi kategórián belül is lehetne hasonló példákat felhozni.

Az első két kategória feladataira az jellemző, hogy igen alaposan kihasználják az indexes változók technikáját. Itt sok számadaton kell egyszerre dolgozni, és a programozási nehézséget általában a megfelelő számadatoknak a memóriából való előhívása okozza. Maguk a végzendő műveletek többnyire igen egyszerűek.

A harmadik kategóriába tartozó feladatok esetén a kiszámítandó formulák bonyolultabbak, az egyszerre fel-

* „Mesterséges" vagy „programozott" intelligenciáról akkor beszélünk, ha egy elektronikus számológéppel az emberi tudat különböző funkcióit imitáltatjuk, mint például tanulás, alakzatok felismerése, környezethez való alkalmazkodás stb.

** A gépi szabályozás, ill. vezérlés feladataiban az elektronikus számológép valamely gép, vagy berendezés működését vezérli, irányítja (pl. egy gyárban egy hengersort szabályoz, repülőgépet vezet stb.).

használt számadatok mennyisége viszont kisebb. Ezeket a feladatokat inkább a skaláris változók technikája jellemzi.

A negyedik kategória feladataira az jellemző, hogy a műveleteket elsősorban nem számokon, hanem egyéb jeleken, jelsorozatokon kell végezni: szövegeken, valamilyen programozási nyelven írt programokon stb.

Ebben a pontban példákat fogunk bemutatni az első három kategóriába tartozó feladatokra. A negyedik kategóriába sorolt feladatok programjait általában nem célszerű — sok esetben nem is lehetséges — ALGOL-ban írni, ezért ilyen példát nem közlünk. A negyedik kategóriába tartozó programok írására az ALGOL ui. a következő ok miatt nem alkalmas. Nyilvánvaló, hogy egy olyan programnak, amely nem számjellegű jelsorozatokon, hanem szövegeken vagy egyéb információn operál, olyan változói kell hogy legyenek, amelyek értékként felvehetnek számoktól vagy logikai értékektől különböző jelsorozatokat is. Egy formulafordító programban például olyan változók is szerepelhetnek, amelyek aktuális értékként a lefordítandó program egy-egy utasítását vagy utasításának egy részét veszik fel. Mivel az ALGOL szintaxisa és szemantikája csak szám- és logikai változók szerepeltetését engedi meg, az ilyen jelsorozatokon operáló programok írására ALGOL-ban nincs mód, vagy legalábbis a program megírását a fentiekkel kapcsolatos nehézségek nagyon kényelmetlenné teszik.

Az ALGOL elsősorban tudományos és műszaki feladatok programjainak írására alkalmas. Az irodai ügyvitel gépesitésének tipikus feladataira a matematikusok külön programozási nyelvet dolgoztak ki, a COBOL-t*, amely az ALGOL-hoz hasonlóan, szintén igen elterjedt és világszerte használják. A folyóiratokban megjelentek az elmúlt években javaslatok különböző egyéb szakterületek „célnyelveinek" kidolgozására is, azonban e nyelvek nem terjedtek el szélesebb körben.

• COBOL = COmmon Business Oriented Language, általános ügyvitelgépesitési célzatú programozási nyelv. Lásd pl. [9].

A következőkben néhány példát adunk adatrendezési feladatokra.

*Példák*

1. Vektor maximális abszolút értékű elemének* megkeresése. Legyen a $v$ vektor $N$ elemű, és keressük meg $v$ maximális abszolút értékű elemét. A maximális elem aktuális értékét adjuk a *maxv* nevű valós változónak, indexének aktuális értékét az *indmax* nevű egész változónak!

A program:

```
maxv := 0;                                                    (3.69)
for i := 1 step 1 until N do if abs (v[i]) ≧ abs (maxv) then
   begin maxv := v[i];   indmax := i end;   {folytatás}
```

2. Vektor elemeinek nagyság szerinti rendezése. A $v$ vektor legyen $N$ elemű, és tegyük fel, hogy $v$ elemeit növekvő sorrendbe kell rendeznünk. Eljárhatunk a következőképpen:

*a)* A vektor $v_i$, $v_{i+1}$, ..., $v_N$ elemei közül kiválasztjuk a legkisebbet; tegyük fel, hogy ez $v_{k_i}$.

*b)* Ha $k_i \neq i$ (ez esetben biztosan $k_i > i$), akkor felcseréljük a $v_i$ vektorelemet $v_{k_i}$-vel.

*c)* Az *a)* és *b)* alattiakat megismételjük $i = 1, 2, \ldots, N-1$-re. Az eljárás során $v$ elemei növekvő sorrendbe rendeződnek.

A program:

```
for i := 1 step 1 until N - 1 do begin
m := v[i];
for j := i + 1 step 1 until N do if v[j] < m then                (3.70)
begin m := v[j];   k := j end;
if k ≠ i then begin v[k] := v[i]; v[i] := m end end; {folytatás}
```

* Az ALGOL 60 nyelvvel kapcsolatban mindig a tömbök *elemeiről* szokás beszélni. Ezért a továbbiakban ehhez tartjuk magunkat, és a közönséges matematikában szokásos egyéb elnevezések (koordináta, komponens, összetevő stb.) használatától eltekintünk.

A (3.70) program lényegében a (3.69) programnak továbbfejlesztése; vegyük észre, hogy megírásánál nem kellett egyebet tennünk, mint (3.69)-et, némi módosítással, beleépítenünk egy külső ciklusba.

3. Legyen $K$ egy $N$ elemű vektor, amelynek elemei egész számok, és legyenek $I$ és $J$ ugyancsak $N$ elemű, egész típusú vektorok. Írjunk programot, amely a $K$ vektor páros elemeit elhelyezi az $I_1, I_2, \ldots, I_k$ vektorelemekbe, a páratlanokat pedig a $J_1, J_2, \ldots, J_m$ vektorelemekbe. Nyilván $k+m=N$. Mivel nem tudjuk előre, hogy $K$ elemei között hány páros és hány páratlan található, mind az $I$, mind a $J$ vektor számára $N$ rekeszt tartunk fent, de természetesen nem mindet használjuk ki. Éppen a $k$ és $m$ számok mutatják meg, hogy hány páros, ill. hány páratlan elem van; ezt az információt is tároljuk további felhasználás céljából.

A program:

```
k:=m:=0;                                                    (3.71)
for i:=1 step 1 until N do
   begin L:=v[i];   if L=L÷2×2 then begin k:=k+1;
   I[k]:=L end
   else begin m:=m+1;   J[m]:=L end end;   {folytatás}
```

A programban az $L$ egész típusú változó közbülső változó (munkarekesz), melyet a $v[i]$ változó tárolására használunk. (Említettük, hogy egy indexes változóhoz való fordulás több időt vesz igénybe, mint egy index nélkülihez.)

Most vizsgáljunk meg néhány lineáris algebrai és statisztikai jellegű feladatot.

1. Legyen $x$ valós típusú vektor. Tegyük fel, hogy $x$ elemei valamilyen statisztikai sokaság számadatai, és a feladatok az elemek számtani közepének és korrigált empirikus szórásnégyzetének meghatározása. Mint a matematikai

statisztika elemeiből ismeretes, a korrigált empirikus szórásnégyzetet az alábbi képlettel definiálják:

$$S^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2, \tag{3.72}$$

ahol $\bar{x} = \frac{1}{n} \sum_{i=n}^{n} x_i$, a sokaság empirikus közepe. Programot kell írnunk ezeknek a mennyiségeknek a kiszámítására. A program a következő lehet:

```
P:=0;  for i:=1 step 1 until n do P:=P+x[i];
P:=P/n;  S:=0;                                          (3.73)
for i:=1 step 1 until n do S:=S+(x[i]-P)↑2;
S:=S/(n-1);  {folytatás}
```

A program elemzését az Olvasóra bízzuk.

**2.** Mátrix transzponálása. Legyen $A$ egy $N \times N$ típusú mátrix; a feladat a mátrix transzponáltjának előállítása, és elhelyezése az $A$ mátrix eredeti helyén. Mint ismeretes, a transzponált mátrix definíciója a következő:

$$\| A^t_{ik} \| = \| A_{ki} \|$$

Feladatunkat a következő ALGOL-programmal oldhatjuk meg:

```
for i:=1 step 1 until N do
   for k:=i+1 step 1 until N do begin                   (3.74)
      m:=A[i, k];  A[i, k]:=A[k, i];  A[k, i]:=m end;
      {folytatás}
```

A (3.74) program belső ($k$ szerinti) ciklusa $i=N$ esetén, tehát az utolsó $i$ szerinti ciklusban, egyetlenegyszer sem hajtódik végre [lásd a (3.50) definíciós programot!], és ezért a külső ciklusban valójában felesleges felső ciklushatárként $N$-et előírni. Egyenértékű programot kapunk, ha (3.74) $i$ szerinti ciklusára felső ciklushatárként $N-1$-et írunk elő.

**3.** Adott síkbeli pontsokasághoz egyenes illesztése a legkisebb négyzetek módszere segítségével. Legyen adva

egy $x$ és egy $y$ vektor, mindkettő tartalmazzon $N$ elemet. $x_i$-t tekintsük abszcisszának, a hozzárendelt ordináta legyen $y_i$. Meghatározandó annak az egyenesnek $A$ és $B$ együtthatója, amely „legjobban" illeszkedik az adott pontsokasághoz. Legjobban illeszkedőnek azt az egyenest tekintjük, amelynek együtthatóira a

$$Q = \sum_{i=1}^{n} (y_i - Ax_i - B)^2 \tag{3.75}$$

kvadratikus kifejezés minimumot vesz fel. Az $A$ és $B$ együtthatók a következő eljárás segítségével határozhatók meg:

*a)* Képzendők az alábbi szummák:

$$S_1 = \sum_{i=1}^{N} x_i, \quad S_2 = \sum_{i=1}^{N} x_i^2, \quad S_3 = \sum_{i=1}^{N} y_i, \quad S_4 = \sum_{i=1}^{N} x_i y_i.$$

*b)* A kapott szummák segítségével $A$ és $B$ a következőképpen fejezhető ki:

$$A = \frac{S_2 S_3 - S_1 S_4}{D}, \quad B = \frac{N S_4 - S_1 S_3}{D},$$

ahol

$$D = N S_2 - S_1^2.$$

Az $A$ és $B$ együtthatók meghatározása után kiszámítandó a $Q$ mennyiség is.* Amint rövid számolással belátható, ehhez csak egy újabb szummát kell képeznünk, amelynek számítása a többi szumma számításával egyidejűleg végezhető. Jelöljük ezt a szummát $S_5$-tel:

$$S_5 = \sum_{i=1}^{N} y_i^2.$$

$S_5$ ismeretében $Q$ kiszámítható a meglevő mennyiségek felhasználásával:

$$Q = S_5 - 2AS_4 + A^2 S_2 - 2BS_3 + 2ABS_1 + NB^2.$$

* A $Q$ mennyiséget a matematikai statisztikában az ún. $\chi^2$ (olv.: khi-négyzet) próbában használják fel.

Feladatunk megoldásának programja:

$$S1:=S2:=S3:=S4:=S5:=0; \tag{3.76}$$

**for** $i:=1$ **step** 1 **until** $N$ **do begin**

$t:=x[i]$; $u:=y[i]$; $S1:=S1+t$; $S2:=S2+t\uparrow 2$;

$S3:=S3+u$; $S4:=S4+t\times u$; $S5:=S5+u\uparrow 2$ **end**;

$D:=N\times S2-S1\uparrow 2$;

$A:=(S2\times S3-S1\times S4)/D$; $B:=(N\times S4-S1\times S3)/D$;

$Q:=S5-2\times A\times S4+A\uparrow 2\times S2-2\times B\times S3+2\times A\times B\times$

$S1+N\times B\uparrow 2$; {folytatás}

Eddigi programozási ismereteink alapján a (3.76) program jól követhető, csupán két megjegyzést kell tennünk.

1. A programban szereplő $t$ és $u$ változókat a gépidővel való takarékosság céljából vezettük be.

2. A $Q$ kifejezés értékét kiszámíthattuk volna a (3.75) formula közvetlen alkalmazásával is, ehhez azonban $A$ és $B$ kiszámítása után egy újabb **for** ciklust kellett volna írnunk, ami mind térben, mind időben* megnövelte volna a programot. Általában hasznos a formulákat úgy átalakítanunk, hogy amit lehet, meglevő részeredmények felhasználásával szá-

mítsunk ki, ezzel programunk rövidebb és gyorsabb lesz.

**4.** Lineáris algebrai egyenletrendszer megoldásának programja. Írjunk ALGOL-programot a következő $n$-ismeretlenes, inhomogén lineáris algebrai egyenletrendszer megoldására:

$$\begin{aligned} a_{11}x_1+a_{12}x_2+\ldots+a_{1n}x_n &= a_{1,n+1} \qquad (3.77)\\ a_{21}x_1+a_{22}x_2+\ldots+a_{2n}x_n &= a_{2,n+1}\\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\ a_{n1}x_1+a_{n2}x_2+\ldots+a_{nn}x_n &= a_{n,n+1}. \end{aligned}$$

* Egy számológép-program *térbeli* hosszán, vagy kiterjedésén a program által elfoglalt memóriarekeszek számát, *időbeli* kiterjedésén pedig a lefuttatásához szükséges gépidőt értjük (a program fordítására fordított gépidőt ebbe nem szokás beszámítani).

Feltesszük, hogy (3.77) determinánsa, valamint sarokdeterminánsainak

$$a_{11},\quad \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix},\quad \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \quad \text{stb.}$$

sorozata 0-tól különböző. Kikötéseinket teljesítő egyenletrendszerek megoldása az alábbi algoritmus alapján végezhető:

$i = 1, 2, \ldots, n$-re végrehajtjuk a következőket:

*a)* Az $i$-edik sor $i+1$, $i+2$, ..., $n+1$-edik elemét elosztjuk $a_{ii}$-vel;

*b)* A $j$-edik sor ($j = 1, 2, \ldots, i-1, i+1, \ldots, n$) $k$-adik elemében ($k = i+1, i+2, \ldots, n+1$) kivonjuk az $i$-edik sor $k$-adik elemének $a_{ji}$-szeresét.

A fenti eljárás $n$-szeri ismétlése után az egyenletrendszer megoldása a szabad tagok helyén fog képződni, tehát az $a_{1,n+1}$, $a_{2,n+1}$, ..., $a_{n,n+1}$ mátrixelemek helyén. A leírt algoritmust Gauss—Jordan-eliminációnak nevezik az irodalomban. Az a feltevés, hogy a sarokdeterminánsok sorozata 0-tól különböző, azt biztosítja, hogy a fenti eljárás folyamán $a_{ii}$ sohasem váljék zérussá, és az algoritmus kisebb módosításával elejthető.

A leírt algoritmusnak a következő ALGOL-program felel meg:

```
for i:= 1 step 1 until n do begin                                   (3.78)
   m:= 1/a[i, i];
   for k:= i+1 step 1 until n+1 do a[i, k]:= m × a[i, k];
   for j:= 1 step 1 until n do if j≠i then begin
      T:= a[j, i];  for k:= i+1 step 1 until n+1 do
      a[j, k]:= a[j, k] - T× a[i, k] end
end;  {folytatás}
```

A (3.78) program egyetlen ciklusutasítás, a ciklusváltozó $i$. A ciklus magja összetett utasítás, melynek utasí-

tásai rendre végrehajtják a leírt algoritmus *a)*, ill. *b)* pontját. Az első utasítás az $a_{ii}$ elem reciprokát képezi, a következő **for** ciklus ($k$ ciklusváltozóval) végigszorozza az $i$-edik sort $a_{il}$ reciprokával. Ezt követi egy kétszeresen egymásba skatulyázott hurok, amely megfelel az algoritmus *b)* pontjának, és a mátrixelemeknek az algoritmus szerinti összevonását végezteti el.

Végül tekintsünk néhány egyéb jellegű feladatot:

**1.** Többváltozós függvény kiszámítása. Készítsünk programot az alábbi háromváltozós függvény kiszámítására:

$$f(x, y, z) = \frac{\sqrt{x + y^2 + z^2 + e^{\frac{x^2 + y^3 + z^3}{3600}}}}{\sin^2(yz) + \ln\left(1 + \frac{1}{z^2}\right)}. \tag{3.79}$$

A (3.79) képletet a következő értékekre kell kiszámítani: $x = 1$-től 0.1 lépésközönként 10-ig, 10-től 0.05 lépésközönként 12-ig, 12-től 0.01 lépésközönként 15-ig; $y = 1$-től 0.2 lépésközönként 20-ig, 20-tól 0.1 lépésközönként 30-ig, 30-tól 0.05 lépésközönként 35-ig; $z = 1$-től 0.1 lépésközönként 20-ig, 20-tól 0.05 lépésközönként 30-ig, 30-tól 0.001 lépésközönként 35-ig. Ilyen fajta függvénytabellázások megadására a numerikus matematikában a következő rövidített jelölésmód alkalmazása szokásos:

| | | |
|---|---|---|
| $x = 1\ (0.1)\ 10,$ | $10\ (0.05)\ 12,$ | $12\ (0.01)\ 15$ |
| $y = 1\ (0.2)\ 20,$ | $20\ (0.1)\ 30,$ | $30\ (0.05)\ 35$ |
| $z = 1\ (0.1)\ 20,$ | $20\ (0.05)\ 30,$ | $30\ (0.001)\ 35$ |

A feladat megoldására az általános típusú ciklusutasítást jól tudjuk alkalmazni, amely lényegében a feladat megadásához használt táblázatnak ALGOL-ba való átírása. A függvényértéket a független változók minden szóba jövő kombinációjára ki kell számítani. A kombinációk száma $431 \times 291 \times 582 = 72\,995\,022$. Ekkora tömegű számadatot egyetlen gép sem tud belső memóriájában tárolni; ezért programunkat annak feltételezésével írjuk meg, hogy a

függvényértékkel a programnak egy más része még valamit csinál, pl. kinyomtatja, vagy egyéb módon felhasználja. Ezt a programrészt a kapcsos zárójelbe írt „nyomtatás" szóval szimbolizáljuk*. A feladat megoldásának programja a következő lehet:

```
for z:=1 step 0.1 until 20, 20.05 step 0.05 until 30, 30.001
step 0.001 until 35 do begin
   m1:=ln(1+1/z↑2);
   for y:=1 step 0.02 until 20, 20.1 step 0.1 until 30, 30.05
   step 0.05 until 35 do begin
      m2:=y↑2+z↑2;  m3:=sin(y×z)↑2;
      for x:=1 step 0.1 until 10, 10.05 step 0.05 until 12,
      12.01 step 0.01 until 15 do begin
         f:=sqrt(x+m2+exp((x↑2+m2)/3600))/(m3+m1);
      {nyomtatás};
end end end;  {folytatás}                                    (3.80)
```

A (3.80) program háromszorosan egymásba skatulyázott ciklusutasítás, amelynél az általános típusú **for** ciklus lehetőségét használtuk ki. A lefutási idő rövidítése érdekében úgy állítottuk össze, hogy a gép ne számolja többszörösen azt, amit nem feltétlenül szükséges. Így például a logaritmus alatti mennyiséget, amely csak $z$-től függ, kiemeltük az $y$ és $x$ szerinti ciklus elé. Hasonlóképpen, a *sin* alatti mennyiség nem függ $x$-től, ezért azt az $x$ szerinti cikluson kívül számoljuk. Különösen nagy nyereség a gépidő szempontjából, ha sikerül néhány standard függvényhez való fordulást megtakarítani, mert ezeknek a kiszámítása általában lényegesen tovább tart, mint az egyéb műveletek elvégzése.

A program térben és időben való rövidítése általában ellentmondó feladatok. A térben való rövidítésre csak akkor

* Gyakorlati feladatok kapcsán általában nincs értelme 73 millió számot tartalmazó táblázatokat kinyomtatni, mert azok gyakorlatilag teljesen kezelhetetlenek, és általában feleslegesek is. Sokkal helyesebb ehelyett magával a géppel végeztetni el azt, amit a táblázat használatával a megrendelő végezne el, pl. optimumok keresését stb.

érdemes törekedni, ha kétséges, hogy különben a program elfér a memóriában. Egyébként a gépidő javára döntünk.

**2.** Transzcendens egyenlet megoldása iterációval. Írjunk ALGOL-programot az alábbi transzcendens egyenlet megoldására:

$$e^{\alpha x-\beta}=\frac{A}{x}+\frac{B}{x^2}+C. \tag{3.81}$$

Iterációs formula elérése céljából a bal oldalból kifejezzük $x$-et, és az egyenletet $x=f(x)$ alakra hozzuk:

$$x=\frac{1}{\alpha}\left\{\ln\left(\frac{A}{x}+\frac{B}{x^2}+C\right)+\beta\right\}.$$

Iterációs formulát úgy kapunk, ha a bal oldali $x$ helyébe $x_{k+1}$-et, a jobb oldali $x$-ek helyébe $x_k$-t írunk:

$$x_{k+1}=\frac{1}{\alpha}\left\{\ln\left(\frac{A}{x_k}+\frac{B}{x_k^2}+C\right)+\beta\right\}. \tag{3.82}$$

Feltesszük, hogy $A$, $B$, $C$, $\alpha$ és $\beta$ aktuális értékei olyanok, hogy a (3.82) iteráció konvergens. Ilyen értékek léteznek. Az iterációt a következő ALGOL-programmal írhatjuk le:

```
t := x0;
for x := (ln (A/t + B/t↑2 + C) + beta)/alfa                    (3.83)
while abs (t − x) >10 − 4 do t := x;
```

Itt $x0$ az iteráció kezdőértéke. A program az egyenletet $10^{-4}$ abszolút pontosságig oldja meg.

**3.** Paraméteres integrál kiszámítása, numerikus integrálással. Tegyük fel, hogy ki kell számíttatnunk elektronikus számológépen a következő kétparaméteres integrált:

$$I(a,b)=\int_0^1 \frac{\sqrt{1+ax^2+bx^3}}{\sqrt{a^2+x^2}+\sqrt[3]{b^3+x^3}}\,dx, \tag{3.84}$$

mégpedig az alábbi paraméterértékekre:

$$a = 1\,(0.1)\,2$$

$$b = 1\,(0.1)\,2.$$

Feltesszük, hogy előzetes numerikus vizsgálatok szerint az integrál kiszámítása a trapézformula elegendő pontosságot ad, ha a [0, 1] intervallumot száz részre osztjuk. Mint a numerikus analízis elemeiből ismeretes, a trapézformula a következő:

$$\int_p^q f(x)\,dx \approx \frac{h}{2}\left[f(p) + 2\sum_{i=1}^{n-1}\left[f_i + f(q)\right],\right] \tag{3.85}$$

ahol $h=(q-p)/n$ és $f_i = f(p+ih)$. Esetünkben $p=0$, $q=1$ és $n=100$.

A (3.84) integrálban az integrandus a 0 helyen

$$f(0) = \frac{1}{a+b},$$

az $x=1$ helyen pedig:

$$f(1) = \frac{\sqrt{1+a+b}}{\sqrt{a^2+1} + \sqrt{b^3+1}}.$$

Maga a program a következő:

```
for a:= 1 step 0.1 until 2 do
for b:= 1 step 0.1 until 2 do begin

    S:= 1/(a + b)+ sqrt(1 + a + b)/(sqrt(a↑2 + 1)+(b↑3 + 1)
    ↑0.333333333);
for x:=0.01 step 0.01 until 0.99 do
    S:= S + 2 × sqrt(1 + a × x↑2 + b × x↑3)/(sqrt(a↑2 + x↑2)+
(b↑3 + x↑3)↑0.333333333);
S:=0.005 × S;   {nyomtatás} end;   {folytatás}
```

A (3.80) programhoz hasonlóan, itt is szimbolikusan kijelöltünk egy nyomtatási utasítást, amely az integrál értékét nyomtattatja ki, esetleg az *a* és *b* paraméterek értékével együtt.

## 9. Gyakorlatok

**1.** Legyenek *u* és *v* komplex számok, melyeknek valós részei *x*1, illetve *x*2, képzetes részei *y*1, illetve *y*2. A meghatározandó komplex szám valós része legyen *x*3, képzetes része pedig *y*3. Írjunk ALGOL-programokat, amelyek

*a)* a $w = u \times v$ szorzatot,

*b)* a $w = u/v$ hányadost határozzák meg!

**2.** Írjunk ALGOL-programot a komplex négyzetgyökvonásra! Legyen a *w* komplex szám valós része *x*, képzetes része *y*; a $t = \sqrt{w}$ komplex szám valós része *u*, képzetes része *v*.

**3.** Készitsünk ALGOL-programot az *x* valós és *y* képzetes résszel megadott komplex szám abszolút értékének és fázisszögének meghatározására. Jelölje arctan *z* a *z* szám arkusztangensének $-\pi/2$ és $+\pi/2$ közé eső főértékét. Ekkor a komplex szám $\varphi$ fázisszögére a következő érvényes:

$$\varphi = \begin{cases} \arctan\dfrac{y}{x}, & \text{ha} \quad x>0 \quad \text{és} \quad y \geqq 0 \\ \arctan\dfrac{y}{x}+\pi, & \text{ha} \quad x<0 \\ \arctan\dfrac{y}{x}+2\pi, & \text{ha} \quad x>0 \quad \text{és} \quad y<0 \\ \dfrac{\pi}{2}, & \text{ha} \quad x=0 \quad \text{és} \quad y>0 \\ \dfrac{3\pi}{2}, & \text{ha} \quad x=0 \quad \text{és} \quad y<0 \\ 0, & \text{ha} \quad x=0 \quad \text{és} \quad y=0 \end{cases}$$

A komplex szám abszolút értékét a valós és képzetes rész négyzetösszegéből vont négyzetgyök adja.

**4.** Tipikus ügyvitelgépesitési jellegű feladat a következő. Legyen *R* egy *N* elemű vektor, melynek elemei jelentsenek raktárkészleteket: minden elem valamely áruféleségnek a raktárban rendelkezésre álló

mennyisége. Az árukat sorszámozzuk 1-től $N$-ig. A raktárba minden nap beszállítanak bizonyos árukat, és minden nap van kiszállítás is. A kiszállítást két vektor segítségével jellemezzük; az első — valós típusú — a kiszállított áruk mennyiségét tartalmazza, a másik — egész típusú — elemei a megfelelő áruféleség sorszámát mutatják. Jelöljük az első vektort $K$-val, a másodikat $I$-vel. Ha az $I$ vektor $j$-edik eleme $I_j = s$, akkor ez azt jelenti, hogy az $s$-edik áruféleségből $K_j$ mennyiséget szállítottak el. Hasonlóan, két vektor segítségével adjuk meg a raktárba való beszállítást is: a $B$ vektor adja meg a beszállított áruk mennyiségét, a $J$ vektor pedig a beszállított áruféleségek sorszámát. Feltesszük, hogy a beszállítás időben mindig megelőzi a kiszállítást, pl. a beszállítások reggel 8 és 10 óra között, a kiszállítások pedig 10 óra után történnek. Írjunk ALGOL-programot, amely a $B$, $K$, $R$, $I$ és $J$ vektorok alapján megadja a raktárkészleteket. Definiálunk most további három vektort: $U$-t, $V$-t és $Z$-t. $Z$ ugyancsak $N$ elemet tartalmaz, és elemei egy bizonyos minimális árukészletet jelentenek, amennyinek biztonsági tartalékként mindig lennie kell a raktárban. Ha elszállítás után a $k$-adik áruféleségből $Z_k$-nál kevesebb maradna, akkor a megmaradó mennyiséget át kell írni az $U$ vektor soron következő elemébe, indexét pedig a $V$ vektor soron következő elemébe. Az $U$ és $V$ vektor ily módon vészjelzést ad, ha az árukészlet valamilyen áruféléből fogytán van.

**5.** Legyen adva egy $t$ vektor, amely $n$ elemet tartalmaz, és egy $v$ vektor, amelynek $N$ eleme van. Feltesszük, hogy a $t$ vektor elemei szigorúan monoton növekszenek, azaz

$$t_1 < t_2 < \ldots < t_n.$$

Jelentse $s_i$ $(i = 1, 2, \ldots, n-1)$ a $v$ vektor azon komponenseinek számtani közepét, amelyekre fennáll a

$$t_i < v_k \leqq t_{i+1}$$

egyenlőtlenség. Feltesszük, hogy $v$-nek bármely eleméhez található olyan $t$-beli elempár, amely közrefogja. Írjunk fel programot az $s_i$ mennyiségek kiszámítására!

**6.** A **while** ciklus általánosabb a **step** ... **until** ciklusnál. Mindaz, ami **step** ... **until** ciklussal leírható, könnyen leírható **while** ciklussal is, fordítva azonban nem érvényes a dolog. Írjuk át a (3.48) és (3.49) programokat **while** típusú ciklus alkalmazásával!

**7.** Valaki a négyzetgyökvonásra az alábbi programot írja fel:

```
x := 1;                                                    (3.87)
for x := 0.5 × (x + A/x) while abs (x − A/x) > 10 − 4 do;
```

Helyes-e ez a program szintaktikai és szemantikai szempontból?

8. A gyakorlati programozás során gyakran felmerül függvények nem ekvidisztáns tabellázásának a szükségessége. Az alkalmazandó — változó — lépésköz néha előírható a feladat előzetes numerikus vizsgálata alapján, néha azonban magának a gépnek kell ún. *automatikus lépéskōz-választást* végeznie. Ilyen jellegū feladat a következō: írjunk programot az

$$y = \operatorname{tg} x$$

függvény tabellázására olyan lépéssūrūséggel, hogy a függvény megváltozása lépésközönként ne legyen nagyobb, mint 0.01. A függvénytáblázatot a $[0;\ \pi/2 - 0.0001]$ intervallumban kell elkészítenünk; a függvényértékek az $f$, az abszcisszaértékek az $x$ vektor elemeibe helyezendōk el.

Könnyū meggondolni, hogy kezdeti lépésköznek választhatunk pl. 0.005-öt; ez egészen $\pi/4$-ig megfelel, ezért írhatunk olyan programot, amely a lépéshossz-választó blokkot csak $x > \pi/4$ értékekre helyezi üzembe.

9. Legyen $A$ egy hatméretū tömb. Írjunk olyan ALGOL-programot, amely 0 értéket adja $A$ valamennyi elemének. Ez igen könnyū feladat, ha hatszorosan egymásba skatulyázott **for** ciklus segítségével oldjuk meg. Próbáljunk olyan programot felírni, amelyben legfeljebb kétszeres mélységig egymásba skatulyázott **for** ciklust alkalmazunk.

Ennek a feladatnak az az érdekessége, hogy a helyes program megtalálása esetén az Olvasó olyan programot is fel tud majd írni, amely $n$-szeresen egymásba skatulyázott ciklust tartalmaz, ahol $n$ maga is változó. Ilyen példát a szövegben nem közöltünk, de a gyakorlati programozás során felmerül a szükségessége, pl. $n$-változós függvények szélsōértékének meghatározásában, vagy $n$-szeres integrálok kiszámítására szolgáló standard programok kidolgozásánál.

10. Írjunk fel programot lineáris algebrai egyenletrendszerek megoldására, a következō algoritmus alapján:

*a)* $i = 1, 2, \ldots, n-1$-re végrehajtjuk a következōket:

*a*1) Az $i$-edik sort az $i+1$-edik elemtōl kezdve végigosztjuk $a_{ii}$-vel;

*a*2) A $j$-edik sor ($j = i+1, i+2, \ldots, n$) $k$-adik ($k = i+1, i+2, \ldots, n+1$) eleméből kivonjuk az $i$-edik sor $k$-adik elemének $a_{ji}$-szeresét, és az eredményt visszahelyezzük a $j$-edik sor $k$-adik elemébe;

*b)* Az egyenletrendszer $n$-edik ismeretlenét ezután közvetlenül kiszámíthatjuk:

$$x_n = a_{n,n+1}/a_{nn};$$

*c)* A további ismeretleneket a következőképpen számítjuk ki: $i = n-1, n-2, \ldots, 2, 1$-re végrehajtjuk az alábbiakat:

*c*1) Az

$$S_i = a_{i,i+1}x_{i+1} + a_{i,i+2}x_{i+2} + \ldots + a_{in}x_n$$

szorzatösszeget kivonjuk $b_i$-ből;

*c*2) Az eredményt osztjuk $a_{ii}$-vel.

Az adott algoritmus, melyet az irodalomban Gauss-eliminációnak neveznek, ugyanolyan feltételek mellett alkalmazható, mint a (3.78) programmal kapcsolatban bemutatott Gauss—Jordan-elimináció, és a kiinduló mátrix is azonos módon adandó meg. Az algoritmus némileg bonyolultabb a Gauss—Jordan-elimináció algoritmusánál, viszont gyorsabb, ezért a numerikus analízis gyakorlatában igen kiterjedten alkalmazzák.

**11.** Írjunk ALGOL-programot lineáris algebrai egyenletrendszereknek Gauss—Seidel-módszer szerinti iteratív megoldására. A Gauss—Seidel-iteráció a következő:

*a)* Kiindulunk az egyenletrendszer megoldásának valamely

$$x^{(0)} = \{x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\}$$

nulladik közelítéséből;

*b)* $i = 1, 2, \ldots, n$-re végrehajtjuk a következőket:

*b*1) behelyettesítjük az első egyenletbe az $x^{(0)}$ közelítést. Miután $x^{(0)}$ nem pontos megoldás, az egyenlet általában nem elégül ki, hanem jobb és bal oldala között eltérés lesz. Ezután úgy javítjuk az első ismeretlent, $x_1^{(0)}$-t, hogy az eltérés eltűnjön. A javított $x_1$ értéket jelöljük $x_1^{(1)}$-gyel;

*b*2) Az $x_1^{(1)}$ javított értéket, és az $x^{(0)}$ vektor megmaradó elemeit behelyettesítjük a második egyenletbe. Javítjuk az $x_2$ ismeretlent oly módon, hogy az itt adódott eltérés eltűnjön;

*b*3) Általánosan az *i*-edik egyenletbe behelyettesítjük az előzőleg javított $x_1, x_2, \ldots, x_{i-1}$ ismeretleneket, és a még nem javított $x_i, x_{i+1}, x_{i+2}, \ldots, x_n$ ismeretleneket. Javítjuk az $x_i$ ismeretlent oly módon, hogy az *i*-edik egyenlet pontosan kielégüljön. A javított $x_i$ értéket jelöljük $x_i^{(1)}$-gyel;

*c)* A *b)* pontban leírtak szerint addig ismételjük a kezdeti közelítés javítását, amíg a kapott gyök kellően pontos nem lesz.

Legyen a megoldandó egyenletrendszer a következő:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n - b_1 = 0$$

$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n - b_2 = 0$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n - b_n = 0.$$

Jelöljük az algoritmus *b*3) pontja szerint definiált eltérést (a megoldás *j*-edik közelítésének az *i*-edik egyenletbe való behelyettesítésével adódó hibát) $r_i^{(j)}$-vel. Képletben:

$$r_i^{(j)} = a_{i1}x_1^{(j)} + a_{i2}x_2^{(j)} + \ldots a_{ii}x_i^{(j-1)} + a_{i,i+1}x_{i+1}^{(j-1)} + \ldots$$

$$\ldots + a_{in}x_n^{(j-1)} - b_i. \tag{3.88}$$

Legyen $\Delta x_i^{(j)}$ az a javítás, amit az eltérés eltüntetése érdekében $x_i^{(j)}$-n eszközölnünk kell. Akkor nyilván

$$\Delta x_i^{(j)} = -\frac{r_i^{(j)}}{a_{ii}} \tag{3.89}$$

és

$$x_i^{(j)} = x_i^{(j-1)} + \Delta x_i^{(j)}. \tag{3.90}$$

A feladat a (3.88), (3.89) és (3.90) formulák beprogramozása.

**12.** Írjunk fel programot $n \times n$ típusú négyzetes mátrix invertálására, a Gauss—Jordan-féle eliminációs módszer alapján! A (3.78) ALGOL-programot felhasználhatjuk nem túlságosan lényeges kibővítéssel nem szinguláris négyzetes mátrix inverzének meghatározására. Az algoritmus az $i = 1, 2, \ldots, n$ értékekre a következő:

*a)* A mátrix *i*-edik sorát elosztjuk $a_{ii}$-vel, és az eredményt visszahelyezzük a mátrix *i*-edik sorába. Kivétel az *i*-edik (a fődiagonálisban álló) elem, amelynek helyébe az $a_{ii}$ elem reciprokát írjuk;

*b)* A mátrix *j*-edik ($j = 1, 2, \ldots, i-1, i+1, \ldots, n$) sorából kivonjuk a mátrix *i*-edik sorának $a_{ji}$-szeresét, kivéve az *i*-edik elemet, amelynek helyébe $-a_{ji}/a_{ii}$ írunk;

Az *a*) és *b*) alattiak *n*-szeri ismétlése után megkapjuk a mátrix inverzét, az eredeti mátrix helyén. Ennek a módszernek az az előnye,

hogy „helyben" invertálja a mátrixot, tehát az inverzió nem igényel lényeges számú külön munkarekeszt. A módszer alkalmazhatóságának feltételei azonosak a (3.78) programra vonatkozó feltételekkel.

**13.** Írjunk fel programot $n$-edfokú algebrai egyenlet megoldására a Horner-módszer alapján!

Legyen a megoldandó egyenlet

$$f(x) = \sum_{i=0}^{n} a_i x^{n-i} = 0.$$

Tegyük fel, hogy ismeretes az $x$ megoldás egy $x_0$ közelítése. Kiindulva $x_0$-ból, rendre képezzük a gyök iteráltjait az alábbi iterációs eljárás szerint:

$$\begin{aligned} b_0 &= c_0 = a_0 \qquad \text{bármely } k\text{-ra} \\ b_i^{(k)} &= x_k b_{i-1}^{(k)} + a_i \\ c_i^{(k)} &= x_k c_{i-1}^{(k)} + b_i^{(k)} \end{aligned}$$

A $b_i$ mennyiségek $i = 0, 1, 2, \ldots, n$-re, a $c_i$ mennyiségek pedig $i = 0, 1, 2, \ldots, n-1$-re képezendők. Az ismeretlen gyök következő közelítését az

$$x_{k+1} = x_k - \frac{b_n^{(k)}}{c_{n-1}^{(k)}}$$

képletből határozzuk meg. Az iteráció abbahagyható, ha két egymásutáni közelítés eltérése elegendően kicsiny.

**14.** „Bűvös négyzet"-nek olyan $n \times n$ típusú mátrixot nevezünk, amelynek valamennyi sor- és oszlopösszege egyenlő. Páratlan $n$ esetén a következő algoritmussal képezhetünk bűvös négyzetet:

*a)* Kiindulunk a mátrix első sorának középső eleméből. Középső elem mindig van, mert feltettük, hogy $n$ páratlan;

*b)* A bűvös négyzet elemei az 1, 2, ..., $n^2$ számok, melyeket az alábbi módon helyezünk el:

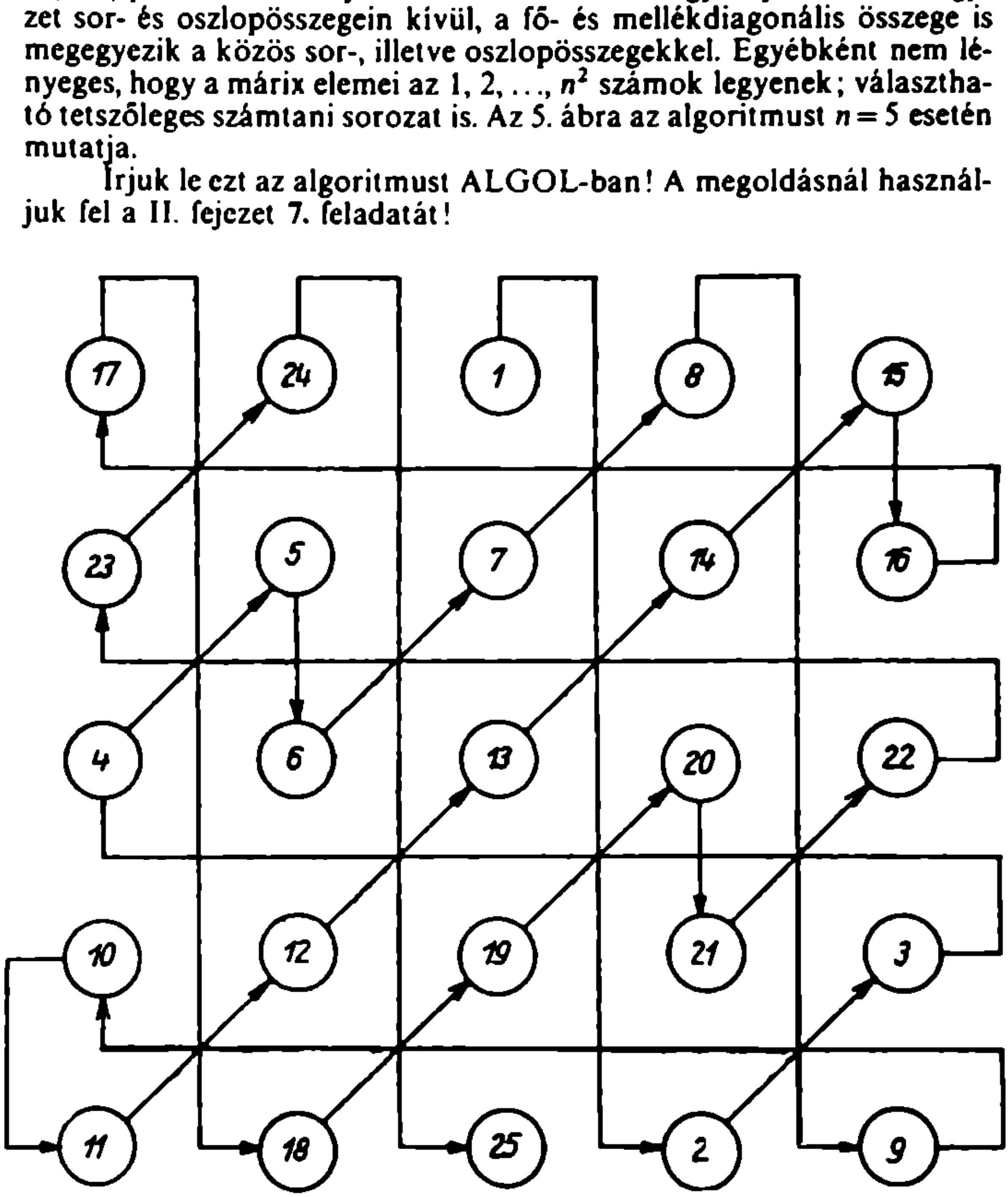
*b*1) Minden olyan esetben, amikor ez lehetséges, a számok a mellékdiagonális irányában növekszenek, azaz ha a $p$ szám a mátrix $i, k$ indexű elemében van, akkor a $p+1$ szám a mátrix $i-1, k+1$ indexű elemébe kerül:

*b*2) Kivételek azok az esetek, amikor a mellékdiagonális szerint soron következő mátrixelem helye nem üres (már foglalt), vagy pedig nem esne a mátrixba (első indexe 0, vagy második indexe $n+1$ lenne). Az első esetben a következő számot az azonos oszlopban eggyel lejjebb álló elem helyére írjuk, tehát az $i+1, k$ indexű elem helyére. A mátrixból való kilépés esetén, a mátrix sor-, illetve oszlopindexeit ciklikusan kell tekinteni, azaz $i = 0$ helyett $i = n$, $k = n+1$ helyett pedig $k = 1$ értendő:

*b*3) A kivételeken belül is speciális eset az $i = 1$, $k = n$ sarokelem, amikor is a mellékdiagonális-szabály szerint az $i = 0$ és $k = n + 1$ indexek egyszerre lépnének fel. Ekkor az eljárást úgy kell folytatni, mintha ez az elem már be volna töltve, azaz ugyanabban az oszlopban maradva, eggyel lejjebb kell menni (az elhelyezendő szám az $i = 2$, $k = n$ indexű elembe kerül).

Be lehet látni, hogy a fenti algoritmus alapján a mátrix „bejárható", vagyis alkalmazása során nem lép fel olyan helyzet, amelyre a *b*1), *b*2), *b*3) pontok valamelyike ne rendelkezne. Az így kapott bűvös négyzet sor- és oszlopösszegein kívül, a fő- és mellékdiagonális összege is megegyezik a közös sor-, illetve oszlopösszegekkel. Egyébként nem lényeges, hogy a márix elemei az 1, 2, ..., $n^2$ számok legyenek; választható tetszőleges számtani sorozat is. Az 5. ábra az algoritmust $n = 5$ esetén mutatja.

Írjuk le ezt az algoritmust ALGOL-ban! A megoldásnál használjuk fel a II. fejezet 7. feladatát!




5. ábra

*Negyedik fejezet*

# A PROGRAM KÍSÉRŐ INFORMÁCIÓI

## 1. Bevezetés

Eddigi ismereteink alapján lényegében bármely algoritmust, amely egyáltalán leírható ALGOL-ban, be tudunk programozni. Tudjuk azonban, hogy azok az ALGOL-programok, amelyekkel a III. fejezetben foglalkoztunk, nem alkotnak zárt egészet, hanem csupán nagyobb programok kiragadott részeinek tekinthetők. Teljességükhöz mindenekelőtt a használt azonosítókról szóló felvilágosítások, az ún. deklarációk hiányoznak. A deklaráció fogalmával már az I. fejezetben megismerkedtünk. Ebben a fejezetben főként az ALGOL nyelvnek a deklarációk megadására, és azok érvényességére vonatkozó szabályait tárgyaljuk, azonban kitérünk a nem deklaráció jellegű kísérő információknak, a *magyarázó megjegyzéseknek (kommentároknak)* használatára és funkciójára is (6. p.). Bevezetünk továbbá néhány olyan fogalmat, amely a deklarálási szabályok ismerete nélkül nem lett volna tárgyalható. Ilyenek a *kapcsolók* és a *helymegjelölő kifejezések* (5. p.). Ezzel lehetővé válik a **go to** utasítás részletes tárgyalása, amelyre a IV. fejezet 3. pontjában nem térhettünk ki.

Amint azt az I. fejezetben említettük, a deklarációk a fordítóprogram számára tartalmaznak információt a programban felhasznált betűjelölésekről, az azonosítókról. A skaláris változók részére a fordítóprogram egy-egy memóriarekeszt* tart fenn, ezért azokat egyszerűen felsoroljuk egy megfelelően megadott listán. Az indexes változók esetében azonban nem ilyen egyszerű a helyzet, ugyanis a fordítóprogramnak azt is tudnia kell, hány azonos nevű

* Esetleg 2--2 vagy 3--3 stb., de mindenesetre minden skaláris változóhoz ugyanannyi memóriarekeszt.

változó tartozik egy-egy tömbhöz. Egy $12 \times 17$ típusú mátrix tárolására például $12 \times 17 = 204$ rekesz kell. A tömbök elemszáma az indexek számától és értékhatáraitól függ, ezért a deklarációnak ezeket is tartalmaznia kell.

Az ALGOL-szerzők a nyelvet úgy igyekeztek definiálni, hogy a lehető legkönnyebb legyen a programozó matematikusok együttműködése, hiszen ez az ALGOL-koncepció egyik alapvető célkitűzése. Ez a körülmény eleve maga után vonja azt a feltételezést, hogy egy ALGOL-programot igen sok esetben nem egyetlen programozó készít el, hanem matematikusok, programozók egy csoportja. A programozók együttműködése vagy úgy történik, hogy egy nagy terjedelmű feladat különböző részeit más és más személy írja meg, vagy pedig úgy, hogy egyes programrészeket változtatás nélkül átvesznek más programokból, programokat publikáló folyóiratokból, ill. programgyűjteményekből.

Magától értetődik, hogy minden önmagában teljes programnak vagy programrésznek saját jelölésrendszere, saját azonosítói vannak. Ha egy programot bele akarunk építeni egy másik programba, csaknem biztosak lehetünk abban, hogy bizonyos gyakori azonosítók mindkét programban szerepelni fognak és mást jelölnek, hacsak előzőleg nem gondoskodunk valamelyik program szisztematikus átjelöléséről. (Például mindkét program használja az *x*, *y*, *a*, *b* stb. betűket.) Ezzel a betűjelöléseknek nemkívánatos konfúziója jön létre.

Elvi akadályba ugyan nem ütközik, de gyakorlatilag igen kellemetlen feladat a programok átjelölés céljából való felülvizsgálata. Gondoljunk arra, hogy ha egy programot pl. öt különböző helyről „ollóznak" össze, mennyire figyelmesen kell azt átvizsgálnia a „főszerkesztőnek", hogy ne maradjon egyetlen ütközés sem a különböző azonositók között.

Az ALGOL nyelv ezt a főszerkesztői feladatot a fordítóprogramra háritja: az ALGOL fordítóprogram maga gondoskodik a különböző jelölésrendszerek összehangolásáról. Az alapgondolat az, hogy az ALGOL-ban a deklarációk érvénye nem az egész programra, hanem annak csak meg-

határozott részére terjed ki, és ezen a programrészen kívül ugyanaz az azonosító felhasználható más célra is.

Ennek további kifejtése céljából először is be kell vezetnünk a *blokk* fogalmát. Az ALGOL nyelvben bármely összetett utasítás **begin**-je után deklarációkat helyezhetünk el. Az olyan összetett utasítást, amelynek **begin**-je után legalább egy deklaráció következik, blokknak nevezzük. A deklaráció érvénye — a megfelelő azonosító ún. *hatásköre* — csak arra a blokkra terjed ki, amelynek kezdetén deklarálták.

Az ALGOL nyelv *blokkstruktúrájának* lényege abban áll, hogy a különböző blokkokban deklarált azonosítók teljesen függetleníthetők egymástól, vagyis ha ugyanaz az azonosító két különböző blokkban van deklarálva, akkor ez már biztosítja azt, hogy ne keveredhessenek össze. Ha tehát a közös azonosítókat tartalmazó programrészeket külön blokkokba helyezzük, ezáltal biztosíthatjuk azok megkülönböztethetőségét. Ez a módszer igen nagy mértékben megkönnyíti a különböző személyek által írt programok összeépítését. A következőkben látni fogjuk, hogy a blokkstruktúra egy sereg szemantikai problémát vet fel, különösen egymásba skatulyázott blokkok esetén.

## 2. Típusdeklarációk

A skaláris változók deklarációja lényegében az azonosítók listaszerű felsorolásából áll. Külön-külön listán soroljuk fel az egész, a valós és a logikai típusú változókat. Az egyes változóazonosítókat vesszővel választjuk el egymástól, és minden egyes változólista után pontosvessző áll. A változók típusának megjelölésére három új alapjelet vezetünk be, ezek az **integer** (egész), **real** (valós) és **Boolean** (logikai) alapjelek.

Ha például egy blokkban nyolc változót kívánunk szerepeltetni, mégpedig az *x*, *y* és *z* valós típusú változókat, az *i*, *j* és *k* egész típusúakat és az *alfa* és *beta* nevű logikaiakat, akkor ennek a blokknak a kezdetén (a blokkot meg-

nyitó **begin** után) a deklarációsorozatot a következőképpen helyezhetjük el:

```
begin real x, y, z;
   integer i, j, k;                                          (4.1)
   Boolean alfa, beta;  {ALGOL-utasítások} end
```

Sem a listákon belüli elemeknek, sem pedig maguknak a listáknak nincs kötött sorrendje. A (4.1) példában is írhattuk volna elsőnek az egész típusú változók listáját, másodiknak a logikaiakét stb. Az sincs kikötve, hogy mindegyik fajta listából csak egyetlen egy lehet egy blokkban: megadható például két valós (**real** alapjellel kezdődő) változólista is. Két dologra azonban ügyelni kell a változók deklarálásával kapcsolatban. Egyrészt, ugyanazt az azonosítót nem szabad kétszer deklarálni, másrészt a blokkban a deklarációs résznek véget vet az első, nem deklaráció jellegű jelsorozat (utasítás), és ezután ugyanazon blokkban újabb deklarációkat már nem szabad írni.

Az ALGOL nyelvben a blokkot megnyitó utasítászárójelet és az utána következő deklarációkat együttesen *blokkfejnek* nevezzük. A blokkfejnek ott van vége, ahol a blokk első ALGOL-utasítása kezdődik.

Az ALGOL-ban az utasítászárójelek — így a blokkok is — tetszőleges mélységben egymásba skatulyázhatók. A következőkben azokat a szemantikai problémákat tárgyaljuk, amelyek az egymásba skatulyázott blokkokkal kapcsolatban merülnek fel.

Legyen például valamely *A* blokkba bele skatulyázva egy *B* blokk. Legyen az *A* blokk fejében két változó deklarálva, *x* és *y*, a *B* blokk fejében pedig két másik változó, *u* és *v*. Valamennyi változó legyen valós típusú. A program szerkezetét az alábbi séma mutatja:

```
A : begin real x, y;   {ALGOL-utasítások};
B : begin real u, v;   {ALGOL-utasítások}
end;   {ALGOL-utasítások};                                   (4.2)
end
```

(4.2)-vel kapcsolatban mindenekelőtt két fogalmat, a *lokális* és a *nem lokális (globális) változó* fogalmát vezetjük be.

Mint látjuk, a (4.2) sémában az *A* blokk teljes egészében tartalmazza a *B* blokkot, *B* része *A*-nak. Az *x* és *y* változók az *A* blokk fejében vannak deklarálva, tehát az 1. p.-ban mondottak szerint érvényességi körük az *A* blokkot bezáró **end**-ig terjed. [Ez a (4.2) sémában a második, „külső" **end**.] Miután az *A* blokk tartalmazza *B*-t, *B* is beletartozik az *x* és *y* változók hatáskörébe. Következésképpen a *B* blokk belsejében szabad olyan ALGOL-utasítást írni, amely *x*-re vagy *y*-ra hivatkozik (például értéket ad *x*-nek vagy *y*-nak), annak ellenére, hogy nem a *B* blokkban vannak deklarálva. Az ilyen változókat *globális változónak* nevezzük *a B blokkra nézve*. Ennek ellentéte a *lokális változó*, amelynek deklarációja ugyanazon blokkban van elhelyezve. Ilyen változók a *B* blokkban *u* és *v*, az *A* blokkban *x* és *y*. Arról, hogy egy változó lokális vagy globális-e, nyilván mindig csak egy meghatározott blokk viszonylatában van értelme beszélni; például *x* és *y* lokális *A*-ra nézve, de globális *B*-re. Miután egy azonosítóra való hivatkozásnak csak hatáskörén belül van értelme, az *u* és *v* változókra hivatkozó utasításokat csakis *B* belsejében használhatunk, *A*-nak *B*-től idegen részében nem.

Miután egy teljes ALGOL-programban csupa deklarált azonosítónak kell szerepelnie, nyilvánvaló, hogy teljes programban nem szerepelhet globális változó. Gyakorlatilag azt mondhatjuk, hogy minden teljes ALGOL-program blokk,* amelyben természetesen lehetnek alacsonyabbrendű alblokkok.

A blokkból való kilépés után a blokk valamennyi lokális változója elveszti értékét. Előfordulhat, hogy a vezérlés kilép egy blokkból egy magasabb rendű blokkba, majd ismét visszakerül ugyanabba az alacsonyabb rendűbe. Minden egyes ilyen ismételt belépéskor az alacsonyabb

* Az ALGOL szintaxisa megengedi, hogy egy ALGOL-program ne blokk, hanem összetett utasítás legyen, de ez csak úgy fordulhat elő, hogy egy vagy több blokkot beskatulyáznak egy utasítás-zárójelpárba. Ilyen konstrukciót azonban a gyakorlati programozásban nem használnak.

rendű blokk valamennyi lokális változója „tiszta lappal" indul, tehát mindaddig nincs értéke, amíg egy értékadó utasítás nem ad neki értéket. Másként fogalmazva, a kilépés után általában nem őrződik meg a lokális változók utolsó értéke. (Lásd azonban a későbbiekben az **own** alapjelről mondottakat.) Az alacsonyabb rendű blokkból magasabb rendűbe való átlépés történhet oly módon, hogy a gép végrehajt egy **go to** utasítást, amelynek rendeltetési címkéje egy magasabb rendű blokkban van, vagy történhet a kilépés a blokkot bezáró **end**-en keresztül is. Mindkét esetben elvész a lokális változók értéke.

Blokkba belépni mindig csak a blokkot megnyitó **begin**-en keresztül szabad, tehát nem lehet **go to** utasítással blokk belsejébe beugrani. Ezzel kapcsolatos részletekre nézve lásd az 5. p.-ot.

Az ALGOL szintaxisa megengedi, hogy ugyanazt az azonosítót két — vagy több — hierarchiailag egymás fölé rendelt blokkban is deklaráljunk. Erre példa a következő:

```
A : begin real x;   x:=2.5;                                    (4.3)
   B : begin integer x;   x:=3;   {ALGOL-utasítások};
      end;
{ALGOL-utasítások}
end
```

Az *A* külső blokkban *x* valós típusú változóként van deklarálva, és ránézve mindjárt értékadás is történik, amelynek következtében felveszi a 2.5 értéket. Ezután a *B* új blokk kezdődik, amelynek fejében *x* újra deklarálva van, de most már mint egész típusú változó. Ebben a blokkban is történik értékadás *x*-re nézve. Kérdés, mi a (4.3) ALGOL-program szemantikája, mi történik az *x* változó értékével?

Az ALGOL 60 szabályai úgy rendelkeznek, hogy a külső blokkban a külső érték, a belsőben a belső érték az érvényes. Tehát a *B* blokknak *x*-re hivatkozó utasításaiban *x* mint egész típusú változó szerepel, melynek aktuális értéke 3 (feltesszük, hogy *B*-ben nincs másik utasítás, amelyik értéket adna *x*-nek). Ugyanakkor az *A* blokknak *B*-től idegen részeiben *x* valós típusú változó, 2.5 aktuális

értékkel. Az újradeklarálással tehát $x$ régi értéke és régi típusa nem vész el véglegesen, hanem csak „látens állapotba" kerül arra az időre, amíg $x$-nek a $B$ blokkbeli lokális deklarációja van érvényben. A $B$ blokkból való kilépés után az $A$ blokkbeli látens érték ismét aktívvá, és az $x = 2.5$ érték ismét hozzáférhetővé válik. Ilyen esetben az $x$ azonosító mind az $A$, mind a $B$ blokkra nézve lokális. A (4.3)-hoz hasonló szituációban sem a belső blokkon kívüli változóhoz nem lehet a belső blokk belsejéből hozzáférni, sem pedig a belső blokkon belüli változóhoz a külső blokkból.

Az előzőkben említettük, hogy a belső blokkból való kilépés után általában nem őrződik meg a lokális változók végértéke, és nem áll rendelkezésre a blokkba való ismételt belépés után. A programozónak azonban módja van külön kikötni bármely lokális változó végértékének megőrzését. Ebből a célból az **own*** speciális alapjelet kell alkalmaznia. Az **own** alapjellel megjelölt változók végértéke a blokkból való kilépés után látens állapotban megmarad, és ismét aktívvá válik, ha a vezérlés újra bekerül ebbe a blokkba.

Az **own** alapjelet mindig a típusdeklaráló alapjel előtt kell elhelyezni. Alkalmazását a következő példán szemléltethetjük:

```
A : begin real x;  {ALGOL-utasítások};
B : begin own real z;  {további deklarációk};  {ALGOL-utasítások};
z := 2.37;  {ALGOL-utasítások};                                    (4.4)
end;
{ALGOL-utasítások};
go to B;
end
```

Itt a (4.4) program belső blokkjában deklaráltunk egy **own** típusú valós változót. Ennek viselkedése a $B$ blokkba való első belépés alkalmával semmiben sem különbözik a

* **own** = saját.

*B* blokk egyéb lokális változóitól (ezek deklarációját itt rövidség kedvéért nem írtuk ki); a *B* blokkba való ismételt belépés után (lásd a **go to** *B* utasítást) azonban a $z = 2.37$ aktuális érték azonnal felhasználható, míg a többi lokális változó értéke elveszett.

Ezen a ponton két megjegyzést kell tennünk. A (4.4)-hez hasonló programokban a belső blokknak olyan kiképzésűnek kell lennie, hogy a blokkba való második belépéskor annak más részei működjenek, mint az első belépéskor. Ugyanis az első belépéskor a *z*-re vonatkozó értékadás előtt nem hajtódhat végre olyan ALGOL-utasítás, amely *z* aktuális értékére hivatkozik, hiszen *z*-nek még nincs értéke. A továbbiakban viszont feltétlenül kell ilyen utasításnak végrehajtódnia, mert ellenkező esetben a korábbi érték megőrzésének semmi értelme sem volna. A *B* blokknak tehát valamiféle elágazást kell tartalmaznia aszerint, hogy a gép először hajtja-e végre a *B* blokkot vagy nem. Jelenlegi programozási ismereteink nem teszik lehetővé ilyen elágazások elegáns programozását, ezért erre példát a későbbiekben fogunk mutatni (lásd 4. p.).

Másik megjegyzésünk, hogy az **own** típusú változókat, magától értetődően, mindig csakis belső blokkokban alkalmazzuk. A legkülső blokkból a gép sohasem lép ki, és abba nem tér vissza, tehát az **own** használata értelmetlen.

Az **own** deklaráció egyébként kombinálható az újradeklarálással, melynek bizonyos értelemben ellentéte. Tekintsük a következő példát:

```
begin real x; {ALGOL-utasítások};
begin own integer x;
{ALGOL-utasítások};                                    (4.5)
end;
{ALGOL-utasítások};
end
```

A (4.5) séma lényegében azonos (4.3)-mal, csak a belső blokk *x* egész változója **own** típusú. Ilyen konstrukció esetén nemcsak a külső blokk változója kerül látens állapotba arra az időre, amíg a belső blokk deklarációja van

érvényben, hanem fordítva is (ez utóbbit biztosítja az **own** kirovás), a belső blokk változója látens állapotba kerül arra az időre, amíg a külső deklaráció van érvényben. Ilyen módon mindkét irányú átmenet esetére biztosítottuk a változók értékének visszanyerhetőségét.

Formális szabály, hogy az **own** típusú változókat a nem **own** típusúaktól mindig elkülönítve kell deklarálni.

Például:

**begin real** *x*, *y*; **own real** *u*, *v*;
**integer** *i*, *j*, *k*; **own integer** *p*; **own Boolean** *kappa*;
{ALGOL-utasítások};
**end**

Ebben a fejezetben tárgyaltunk egy sereg szemantikai szabályt az egymásba skatulyázott blokkok értelmezésével kapcsolatban. Szükségesnek tartjuk ismételten hangsúlyozni, hogy a blokkstruktúra főként nagy volumenű feladatok programjaiban előnyös. Olyan feladatok esetén, amelyekre egy matematikus — más programokból átvett részek felhasználása nélkül — elfogadható időn belül fel tud programot írni, általában nincs értelme több hierarchikusan egymás fölé rendelt blokknak. Éppen ezért a gyakorlati ALGOL-programok jelentős részében a jelen pontban elmondott szemantikai szabályok nagyrészét nem alkalmazzák. Egyszerű esetben az összes változót a program elején deklaráljuk. Ilyenkor az egész program egyetlen blokkból áll. Komplex feladatok programjainak megírásakor azonban az e fejezetben foglaltak hatékonyan alkalmazhatók.

## 3. Tömbdeklarációk

A tömbdeklarációk felvilágosításokat tartalmaznak a fordítóprogram számára a tömb méretszámáról (dimenzió-számáról) és az egyes indexek értékhatárairól. Az indexek számának és értékhatárainak ismeretében a tömb elemeinek száma kiszámítható. A fordítóprogram a tömbdekla-

rációkból tudja meg, mekkora memóriateret kell fenntartania a célprogram indexes változói részére.

A tömbdeklarációk céljára bevezetünk egy újabb ALGOL-alapjelet: **array**.* A tömbdeklaráció egy típusdeklaráló alapjellel (**real**, **integer** vagy **Boolean**) kezdődik. Ezt követi a most bevezetett **array** alapjel, majd a tömb azonosítója. Ezután szögletes zárójelben az egyes indexekre vonatkozó értékhatárok következnek, listaszerű felsorolásban. A tömbdeklarációk megadási módját az alábbi sémával szemléltethetjük:

$$\textbf{real array } M[A1 : B1, A2 : B2, \ldots, An : Bn] \qquad (4.6)$$

Itt $A1$; $B1$, $A2$, $B2$, ..., $An$, $Bn$ a legegyszerűbb esetben egész számok, általában azonban tetszőleges aritmetikai kifejezések. A tömb azonosítója $M$. A szögletes zárójelben álló listát korlátpár-listának nevezzük. Ez annyi elemet tartalmaz, ahány indexe az $M$ tömbhöz tartozó indexes változóknak van. A korlátpár-lista elemei két-két aritmetikai kifejezésből állnak, melyek egymástól kettősponttal vannak elválasztva. A kettőspontoktól balra álló ($A1$, $A2$, ..., $An$) aritmetikai kifejezések a megfelelő index értékének alsó határát, a kettőspontoktól jobbra álló ($B1$, $B2$, ..., $Bn$) aritmetikai kifejezések pedig felső határát definiálják. Az indexek határainak számszerű értékét a korlátpár-listában szereplő aritmetikai kifejezések aktuális értéke határozza meg, ezért az indexhatárok számszerű értéke a program futása közben változhat. Ebben az értelemben az ALGOL-ban „dinamikus indexhatárokról" beszélünk. A tömbdeklarációt bevezető típusdeklaráló alapjel a tömb típusát definiálja (a tömbhöz tartozó valamennyi indexes változó azonos típusú).

Példák tömbdeklarációkra:

$$\textbf{Boolean array } V[1 : 2 \times n \times m + 3 \times k] \qquad (4.7)$$

$$\textbf{real array } Matrix\,[-3:1+n\,|\,2,\ x+3\times y : U \times V] \qquad (4.8)$$

$$\textbf{integer array } I\,[0:n,\ 0:m,\ 0:k] \qquad (4.9)$$

* **array** szó szerint elrendezés. Az ALGOL nyelv terminológiájában a tömb fogalmával azonosítjuk.

A (4.7) tömbdeklarációval egyméretű, logikai típusú tömböt deklaráltunk, melynek indexhatására 1-től a $2nm+3k$ kifejezés aktuális értékéig terjed. A (4.8) tömbdeklaráció valós típusú, kétméretű tömböt definiál, $-3$-tól $1+n^2$-ig, illetve $x+3y$-tól $UV$-ig terjedő indexhatárokkal. A harmadik példában egész típusú tömb deklarációját adtuk; a tömb háromméretű, indexhatárai 0-tól $n$-ig, $m$-ig, illetve $k$-ig terjednek.

Az indexhatárokat meghatározó aritmetikai kifejezésekről három szemantikai szabály rendelkezik:

1. Ha egy indexhatárt definiáló aritmetikai kifejezés valós típusú eredményt állít elő, erről egész típusú eredményre ugyanúgy térünk át, mint az indexes változók indexének kiszámítása során, tehát az értékadó utasításoknál tárgyalt szabály szerint. [Lásd a III. fejezet 2. p. (3.3) képletét, és a III. fejezet 6. p.-ban az index egésszé alakításáról mondottakat.]

2. A korlátpár-listában szereplő aritmetikai kifejezések mindegyikének értékkel kell bírnia, amikor a vezérlés a tömbdeklarációt tartalmazó blokk **begin**-jéhez érkezik. Ebből adódik a fontos szabály, hogy azokat a változókat, amelyek egy tömbdeklaráció aritmetikai kifejezéseiben szerepelnek, nem szabad ugyanebben a blokkban deklarálni, hanem okvetlenül magasabb rendű blokkban. Speciálisan egy teljes ALGOL-program legkülső blokkjában csakis olyan tömbdeklaráció szerepelhet, amelynek valamennyi indexhatára számkonstanssal van megadva. Teljes ALGOL-programnak ugyanis nem lehetnek globális változói.

3. Az alsó indexhatárokat meghatározó aritmetikai kifejezéseknek nem szabad nagyobb aktuális értéket előállítaniuk, mint a felső határokat meghatározó kifejezéseknek, ellenkező esetben a tömbdeklaráció értelmetlen.

A tömbdeklarációkat, akárcsak a skaláris változók deklarációit és minden egyéb deklarációt, mindig valamely blokk fejében adjuk meg. Ha ugyanazon blokkfejben több azonos méretű és azonos indexkorlátokkal biró tömböt kívánunk deklarálni, rövidített írásmóddal élhetünk: elegendő az indexkorlátokat csupán egyszer megadnunk, és ezt meg-

előzően listaszerűen, egymástól vesszővel elválasztva, felsorolnunk azokat a tömbazonosítókat, amelyekre a közös indexhatárok vonatkoznak.

Tegyük fel például, hogy deklarálni kívánunk két $n$ elemű vektort, és három $n \times m$ típusú mátrixot. A vektorok azonosítói legyenek $u$ és $v$, a mátrixoké $P$, $Q$ és $R$. Ekkor felírhatjuk a következő tömbdeklarációkat:

**begin real array** $u, v[1:n], P, Q, R[1:n, 1:m];$ (4.10)

Több tömb egyidejű deklarációja esetén elegendő az **array** alapjelet is csupán egyszer alkalmaznunk az egy csoportban megadott tömbdeklarációk előtt. Különböző típusú tömbök külön csoportban deklarálandók. Tegyük fel például, hogy valamely blokk fejében egy egész típusú vektort ($I$) és két valós típusú mátrixot kívánunk deklarálni. A megfelelő blokkfej a következőképpen írható fel:

```
begin integer array I[1:n];                    (4.11)
   real array A, B[i:k, i-1:k+1];
```

Megjegyezzük még, hogy az **array** alapjel elől elhagyható a **real**: ha az **array** előtt semmiféle típusdeklaráló alapjel nem áll, úgy a tömböt valós típusúnak kell tekinteni. A (4.11) blokkfej például így is írható:

```
begin integer array I[1:n];                    (4.12)
   array A, B[i:k, i-1:k+1];
```

Az egy csoportban megadott tömbdeklarációk után pontosvessző következik, amely a csoport végét jelzi. Az azonos típusú tömböknek egy csoportban való deklarálása csak lehetőség, amelynek segítségével rövidíthetjük az ALGOL-programot, de alkalmazása nem kötelező; deklarálhatunk akár minden tömböt külön is. Például:

```
array u[1:n]; array v[1:n];
array P[1:n, 1:m];
array Q[1:n, 1:m];
```

stb. A blokkfej természetesen annál áttekinthetőbb, minél kevesebb fölösleges alapjel van benne.

Eddigi ismereteink alapján — eltekintve a ki- és beviteli egységekre hivatkozó utasitásoktól — képesek vagyunk teljes ALGOL-programokat felírni.

Annak érdekében, hogy példát adjunk teljes ALGOL-programra, tekintsük ismét a (3.75) formula kapcsán már tárgyalt (lásd III. fejezet 8. p.) feladatot, egyenes illesztését adott pontsokasághoz. A program „operatív" része — (3.76) — változtatás nélkül felhasználható. Szükségesek azonban bizonyos kiegészítések, amelyek e programot „teljessé" teszik.*

A gyakorlatban az ilyen illesztési feladatok sohasem jelentkeznek egyedileg, hanem mindig, vagy legalábbis csaknem mindig, tömegesen. Egyetlen egyenes illesztését nem is érdemes elektronikus számológépen végezni, az kiszámolható asztali számológépen is. Ha reális programot akarunk írni, akkor ezt annak feltételezésével kell tennünk, hogy a gép sorozatban olvas be több adathalmazt, kiszámolja az együtthatókat, majd nyomtat. Ehhez egy kiegészítő programrészre, ún. vezérprogramra van szükség, amely körülveszi és vezérli a (3.76) programot. A vezérprogram tartalmaz beolvasási és nyomtatási utasitásokat, továbbá bizonyos logikai vizsgálatokat, amelyek alapján a gép megállapitja, van-e még hátra további adatsorozat. Ilyen vizsgálat a ciklusutasítások valamelyik fajtájának segitségével szervezhető meg. A vezérprogram tartalmazza a program végét jelző megállást is.

Miután a vezérprogramnak szinte valamennyi utasítása olyan, hogy csak valamely konkrét gépi reprezentánsban lenne pontosan megfogalmazható, azt csak zárójelben, funkcióját meghatározó megjegyzések formájában jelezzük. A programnak egy meghatározott gépre való alkalmazása esetén a csak jelzett utasításokat a gépi reprezentánsnak megfelelő konkrét utasításokkal kell pótolni.

* A könyv hátralevő részében közölt ALGOL-programokról és programsémákról általában már nem kell feltennünk, hogy azok nagyobb programok kiragadott részletei, ezért e programok végén a „folytatás" szót csak ott tüntetjük fel, ahol a program természete megköveteli.

A program:

```
begin integer N;
{N beolvasása};                                                    (4.13)
begin real S1, S2, S3, S4, S5, t, u, A, B, D, Q;   integer i;
array x, y[1:N];
{Program végének vizsgálata}; {x és y vektorok beolvasása};
S1:=S2:=S3:=S4:=S5:=0;
for i:=1 step 1 until N do begin
        t:=x[i];   u:=y[i];   S1:=S1+t;   S2:=S2+t↑2;
        S3:=S3+u;   S4:=S4+t×u;   S5:=S5+u↑2 end;
D:=N×S2-S1↑2;
A:=(S2×S3-S1×S4)/D;   B:=(N×S4-S1×S3)/D;
Q:=S5-2×A×S4+A↑2×S2-2×B×S3+2×A×B×
        S1+N×B↑2;
{A, B és Q kinyomtatása}; {Visszatérés a program végének
vizsgálatára};
{Megállás} end end
```

A (4.13) program két egymásba skatulyázott blokkból áll. A külső blokk egyetlen deklarációt tartalmaz, mégpedig az egész típusú $N$ változó deklarációját. Ezt a változót ui. a tanult szabályok értelmében magasabb rendű blokkban kell deklarálni, mint az $x$ és $y$ vektorokat, mert azoknak a deklarációjában már szerepel $N$. Miután a következő blokk deklarációs részének elérése előtt arról is gondoskodnunk kell, hogy $N$ értékkel bírjon, annak beolvasásával kezdődik a program végrehajtása. A program többi változója tetszés szerint deklarálható akár magasabb rendű, akár alacsonyabb rendű blokkban. Ilyenkor általában az alacsonyabb rendű blokkban való deklarálás mellett szoktunk dönteni, mert bizonyos esetekben ez memóriatakarékosságot eredményezhet. A jelen példában azonban ennek semmiféle jelentősége nincs.

Konkrét gépi reprezentáció esetén minden valószínűség szerint a vezérprogramnak is lesznek saját változói;

a (4.13) programban mi csak azokat a változókat tüntettük fel, amelyeket a program operatív része használ. A zárójeles utasításoknak konkrét utasításokká való átírásával egyidőben a vezérprogram esetleges változóit is deklarálni kell.

A 2. p.-ban a változódeklarációknál ismertetett újradeklarálási szabályok, valamint az **own** típusú változókkal kapcsolatban mondottak értelemszerű változtatásokkal átvivődnek a tömbváltozókra is. Ha egy tömbváltozó azonosítóját valamely alacsonyabb rendű blokkban újradeklarálják, akkor a tömb valamennyi eleme látens állapotba kerül, és a lokális deklaráció érvényének megszűnése után valamennyi tömbelem ismét hozzáférhető.

Ami az **own** típusú tömböket illeti, azokra hasonló a helyzet, mint a skaláris változókra. Az **own** típusú tömb valamennyi eleme megőrzi aktuális értékét a blokkból való kilépés után, az ugyanazon blokkba való legközelebbi belépésig. Az ALGOL-jelentés rendelkezései értelmében egy **own** típusú tömb elemei közül csak azoknak az értéke van definiálva, amelyeknek indexei az utoljára kiszámított indexhatárok közé esnek.

Ez a kirovás némi magyarázatra szorul.

Tudjuk, hogy a blokkfejben elhelyezett tömbdeklarációkban az indexhatárokra általában aritmetikai kifejezések adottak, és ezek aktuális értéke határozza meg a mindenkori numerikus indexhatárokat. E kifejezések aktuális értékét mindig a blokkba való belépéskor számítjuk ki, a kifejezéseket felépítő változóknak blokkon belüli értékváltozásai már nem befolyásolják az indexek értékhatárait. Legyen például az alábbi blokkba való belépés pillanatában $n$ aktuális értéke 6.

```
begin array v[1:n];
{utasítások};
n:=n-1;                                   (4.14)
P:{utasítások};
end
```

A blokknak a *P* címkével kezdődő részében *n* aktuális értéke már nem 6, hanem 5, ennek ellenére a blokknak ebben a részében is szabad a $v[6]$ indexes változóra hivatkoznunk, mert a határkifejezést a blokkban való tartózkodás tartama alatt már nem számítjuk ki mégegyszer, hanem legközelebb majd csak akkor, amikor a vezérlés legközelebb ugyanennek a blokknak a **begin**-jéhez ér. Ezután természetesen már nem lesz értelme a $v[6]$ változóra való hivatkozásnak, továbbá akkor sem lenne értelme, ha az *n* változónak nem a blokk belsejében, hanem azon kívül adnánk új értéket, és ezt követőleg térnénk rá a blokk **begin**-jére.

Közönséges (nem **own** típusú) tömbök esetében ez egyszerű és világos. Bonyolultabb a dolog **own** típusú tömbök esetében. Tekintsük például a következő programsémát:

```
begin integer n;
n:=6;  {utasítások};                                        (4.15)
Q : begin own array v[1:n];  {utasítások};  end;
{utasítások};  n:=n-1;  go to Q;  {utasítások} end
```

Itt felmerül a kérdés, hogy mi a helyzet a $v[6]$ változóval? Ugyanis egyfelől az **own** alapjel azt a követelményt támasztja, hogy a *Q* blokkba való ismételt belépés esetén a *v* tömb valamennyi eleme álljon rendelkezésre, ugyanakkor azonban az indexhatárt meghatározó kifejezés, jelen esetben az *n* változó, értéke időközben megváltozott, és így az **own** kikötés ellentétbe került a tömb deklarációjával, legalábbis a $v[6]$ változóra nézve.

Az ALGOL-jelentés — mint láttuk — úgy rendelkezik, hogy ilyen esetben a deklaráció az erősebb, tehát az **own** kikötés csakis azokra a tömbelemekre érvényes, amelyek az újabb belépés után is megfelelnek a deklarált indexhatároknak.

Az **own** típusú tömbök kérdése egyébként az ALGOL 60 nyelvnek egyik nem teljesen lezárt problémája. A módosított ALGOL-jelentésben utalás van arra, hogy az **own** típusú tömbök szemantikai problémájára a későbbiekben vissza fognak térni. Az **own** típusú tömbök alkalmazását

általában a fordítóprogramok korlátozzák vagy teljesen megtiltják, használatuk igen ritkán szükséges, és ezért hivatkozási nyelven írt programokban is ajánlatos alkalmazásukat elkerülni. A legegyszerűbb megoldás, hogy az **own** típusú tömb helyett a kívánt tömböt egy magasabb rendű blokkban deklaráljuk, amely esetben az globális az alacsonyabb rendű blokkra nézve, így értéke nem vész el az alacsonyabb rendű blokkból való kilépés után. Ugyanez megtehető az **own** típusú skaláris változókkal is.

## 4. Kapcsolók, helymegjelölő kifejezések

A gyakorlati programozásban gyakran szükséges a programok elágaztatása. Logikai feltételek szerint elágaztatásra már igen sok példát láttunk a III. fejezetben: erre a célra a feltételes kifejezéseket és a feltételes utasításokat, „ciklus vége" vizsgálatoknál a ciklusutasításokat használjuk. Sok esetben azonban másfajta elágaztatásra van szükség. Egy logikai feltétel két különböző értéket (**true** vagy **false**) vehet fel, az egész típusú aritmetikai kifejezés (speciális esetben egész típusú változó) értéke viszont végigfuthat az összes egész számon. A logikai feltétel szerinti elágazásban tehát csak két programrész között választhatunk, aszerint, hogy a logikai feltétel **true** vagy **false** értékű-e. Egy egész típusú kifejezés alapján viszont, legalábbis elvben, tetszőlegesen sok programrész közül választhatjuk ki a megfelelőt. Ezeket megszámozzuk az 1, 2, ..., *n* számokkal, és egy egész értékű aritmetikai kifejezés segítségével meghatározzuk a kívánt variáns sorszámát. Az ALGOL-ban az ún. kapcsolók teszik lehetővé a megadott sorszámú programrészre való vezérlésátadást.

A kapcsolóknak azonosítójuk van és deklarálni kell őket. Először a kapcsolódeklarációk szintaxisával foglalkozunk.

A kapcsolódeklarációk mindig a **switch*** alapjellel kezdődnek, ez után áll a kapcsoló azonosítója, melyet a már

* **switch** = kapcsoló, kapcsolj.

ismert := (kettőspont-egyenlő) alapjel követ. A kapcsolódeklaráció egy címkelistával* fejeződik be; a lista egyes elemei egymástól vesszővel vannak elválasztva. A deklaráció után pontosvessző (;) következik. Példa kapcsolódeklarációra:

**switch** $S := P, Q, 27, 32, F$; (4.16)

Itt $S$ a kapcsoló azonosítója és $P$, $Q$, 27, 32 és $F$ a programban szereplő címkék.

A programban az $S$ kapcsolóra oly módon hivatkozhatunk, hogy felírunk egy vezérlésátadó (**go to**) utasítást, amely után azonban most nem címke következik, mint az eddig vizsgált vezérlésátadó utasítások esetében, hanem az $S$ kapcsolóazonosító és egy aritmetikai kifejezés szögletes zárójelben. Például:

**go to** $S[i+j]$; (4.17)

A (4.17) utasítás végrehajtása úgy történik, hogy kiszámítjuk a szögletes zárójelben álló aritmetikai kifejezés aktuális értékét. Ezt követőleg a gép megkeresi az $S$ kapcsoló deklarációjában az annyiadik elemet, amennyi az aritmetikai kifejezés aktuális értéke, és végrehajt egy vezérlésátadást erre a kapcsolólista-elemre (címkére).

Legyen például a (4.17) vezérlésátadó utasításban $i$ aktuális értéke 2, $j$ aktuális értéke 3. Ebben az esetben a szögletes zárójelben szereplő aritmetikai kifejezés aktuális értéke 5. Következésképpen a gép megkeresi a (4.16) kapcsolódeklarációban az ötödik kapcsolólista-elemet, ez az elem $F$. Tehát az $i$ és $j$ változók megadott aktuális értékei mellett a (4.17) utasítás hatása a **go to** $F$ utasítással egyenértékű. Látható, hogy a kapcsolók valóban alkalmasak a programnak akárhányfelé ágaztatására egy adott helyen. A **go to** utasításban szereplő aritmetikai kifejezés értékének szabályozásával tudjuk a vezérlésátadás rendeltetési címkéjét változtatni.

A (4.16) kapcsolódeklaráció listáján csakis akkor lehet a (4.17) utasításnak megfelelő címkét találni, ha az arit-

* Később látni fogjuk, hogy a lista elemei nemcsak címkék lehetnek.

metikai kifejezés értéke 1, 2, 3, 4 vagy 5. Ellenkező esetben ugyanis nincs értelme a kapcsolólistának „annyiadik" eleméről beszélni, amennyi az említett aritmetikai kifejezés aktuális értéke. Az ALGOL 60 szemantikai szabályai úgy rendelkeznek, hogy ekkor a vezérlésátadó utasítás egyénértékű egy üres utasítással.

Ha a vezérlésátadó utasításban szereplő aritmetikai kifejezés valós típusú eredményt állít elő, ugyanúgy kell egész típusú eredményre áttérni, mint indexes változók esetében. Általában a kapcsolóra hivatkozó ún. kapcsolókifejezések és az egyindexes változók között bizonyos analógia áll fenn, ami nemcsak a formai hasonlóságban, hanem a szemantikai szabályokban is megnyilvánul.

Lássunk most néhány példát kapcsolók alkalmazására!

1. Teljes ALGOL-program, paraméteres integrál kiszámítására. Tegyük fel, hogy ki kell számítani az

$$I(y) = \int_0^{q(y)} \frac{dx}{\sqrt{1 + p(y) + x^4}}$$

paraméteres integrált, ahol

$$p(y) = \sqrt{1 + 1.14y + 2.877y^2}$$

és

$$q(y) = 1 - 0.076y.$$

Az integrál értékét tabellázni kell a $0 \leqq y \leqq 1$ intervallumban, 0.02 lépésközzel.

Az integrálást végezzük a Simpson-formula segítségével! Tegyük fel, hogy az előzetes numerikus vizsgálatok szerint az intervallum száz részre való felosztása minden esetben elegendő pontosságot biztosít. Az integrálást fix lépésközzel végezzük.

Mint a numerikus analízisből ismeretes, a Simpson-formula a következő:

$$\int_a^b f(x)\,dx \approx \frac{h}{3} \sum_{i=0}^{n} c_i f_i, \qquad (4.18)$$

ahol

$$h = \frac{b-a}{n}, \quad f_i = f(a+ih),$$

$n$ páros szám, a $c_i$ együtthatók pedig a következők:

$$c_0 = c_n = 1,$$

$$c_{2j-1} = 4 \quad \left(j = 1, 2, \ldots, \frac{n}{2}\right),$$

$$c_{2j} = 2 \quad \left(j = 1, 2, \ldots, \frac{n}{2} - 1\right).$$

A Simpson-formuláról megjegyezzük, hogy bár alig valamivel bonyolultabb, mint a trapézformula [lásd (3.85)], sok esetben annál lényegesen pontosabb, ezért a numerikus integrálás gyakorlatában kiterjedten használják.

A program a következő:

```
begin real y, p, q, x, delta, I, k;                                (4.19)
   switch S:= L1, L2, L3;
   for y:=0 step 0.02 until 1 do begin
   q: = 1-0.076 × y;   p: = sqrt (1 + 1.14 × y + 2.877 × y↑2);
   delta :=q/100;   I:=0;   k:=0.5;
   for x:=0 step delta until q do begin
      I: = 2 × k/(1 + p + x↑4) + I;   go to S[if x + delta ≧ q then
            3 else k];
      L1 : k:=2;   go to P;
      L2 : k:=1;   go to P;
      L3 : k:=0.5;
   P : end;   I: = delta/3 × I;   {I nyomtatása} end end
```

Az $S$ kapcsoló a függvényértékeknek a megfelelő súllyal (együtthatóval) való szorzását biztosítja. A program összeállításakor ügyelnünk kellett arra is, hogy az első és az utolsó függvényérték helyes súllyal legyen megszorozva.

Az első és utolsó ciklust egyelőre nem tekintve, vizsgáljuk meg, mit hajt végre a program a határozottan 0 és $q$ közé eső $x$ értékekre! A programnak két fő ága van, melyek az $L1$ és az $L2$ címkével kezdődnek. Az $x$ szerinti ciklusok folyamán egyszer az egyik, egyszer a másik ágat kell bejárni, attól függően, hogy a soron következő függvényértéket 2 vagy 4 súllyal kell-e venni. Mindkét programág tartalmaz egy-egy utasítást, amely gondoskodik arról, hogy a legközelebbi ciklusmenetben a másik ág kerüljön bejárásra. Úgy mondhatjuk, hogy a gép „a saját háta mögött" átállít egy váltót, ezáltal lezárva az egyik ágat és kinyitva a másikat. Maga a váltó nem egyéb, mint az $S$ kapcsolóra hivatkozó vezérlésátadó utasítás, melynek aritmetikai kifejezése az $x$ szerinti ciklus belsejében mindig $k$ aktuális értékével határozódik meg. (A logikai feltétel csak a legutolsó ciklusban lesz **true** értékű.) A ciklus belsejében tehát a vezérlés mindig vagy $L1$-re, vagy $L2$-re adódik át, aszerint, hogy $k$ értéke 1 vagy 2.

A ciklus elején és végén külön biztosítjuk, hogy a gép a $c=1$ súlyt vegye figyelembe. A ciklus elején ezt a $k$ változó előzetes beállításával ($k:=0.5$) érjük el, a végén pedig a kapcsolókifejezésben szereplő logikai feltétel **true** értékűvé válik, és a program harmadik ága teszi $k$-t 0.5-tel egyenlővé.

A fentiek után már érthető az a kisebb programozási fogás, hogy nem deklaráltunk külön egy $c$ változót, amely a függvényérték szorzófaktorát tartalmazná, hanem kihasználtuk az *ad hoc* lehetőséget, hogy a szorzófaktor mindig a kapcsolóindex kétszerese, tehát $2 \times k$-val szorzunk. Ez szükségessé teszi, hogy $k$-t valós típusúként deklaráljuk, ugyanis a ciklus elején és végén $k$-nak a 0.5 tört értéket kell adnunk. Ezzel mindjárt láttunk egy példát valós típusú aritmetikai kifejezésnek kapcsolóindexként való alkalmazására. Kihasználjuk a programban azt a tényt is, hogy 0 kapcsolóindex esetén a vezérlésátadó utasítás „üres"-nek tekintendő. Ez a helyzet az első ciklusban.

**2.** Szakaszonként változó képlettel definiált függvény kiszámítása. A kapcsolók másik igen gyakori alkalmazási területe a szakaszonként változó képlettel definiált függ-

vények kiszámításának programozása. Tekintsük például a következő függvénykapcsolatot:

$$F(u,t)=\begin{cases} 0, & \text{ha} \quad u<-s \\ u+\dfrac{u^2}{2s}+G_1, & \text{ha} \quad -s\leqq u<-\dfrac{2Rt}{l} \\ u-\dfrac{la}{3t}\left(u-\dfrac{bt}{l}\right)^3+G_2, & \text{ha} \quad -\dfrac{2Rt}{l}\leqq u<0 \\ u-\dfrac{la}{3t}\left(u+\dfrac{bt}{l}\right)^3+G_3, & \text{ha} \quad 0\leqq u<\dfrac{2Rt}{l} \\ u-\dfrac{u^2}{2s}+G_4, & \text{ha} \quad \dfrac{2Rt}{l}\leqq u<s \\ G_5, & \text{ha} \quad u\geqq s. \end{cases}$$

A szereplő betűjelölések jelentése a következő:

$$a=\frac{\omega}{4d}, \quad b=2R, \quad G_1=G_3=\frac{s}{2},$$

$$G_2=G_4=\frac{s}{2}-\frac{4}{3}\frac{R^3\omega}{d}\left(\frac{t}{l}\right)^2, \quad G_5=s-\frac{4}{3}\frac{R^3\omega}{d}\left(\frac{t}{l}\right)^2.$$

Olyan programot kell felírnunk, amely eldönti az adott $u$ értékről, hogy a megadott hat egyenlőtlenség közül melyiknek tesz eleget, és ennek megfelelően kiszámíttatja a megfelelő formulát. Ez a programunk — ellentétben a (4.19) programmal — nem teljes, mert tartalmaz egy sereg globális változót. A megfelelő programágra való rátérést ismét egy kapcsoló megfelelő beállításával biztosítjuk.

A program:

```
begin real v, a, b, alfa, ro, gamma; integer i;                    (4.20)
switch A:=A1, A2, A3, A4, A5, A6;
v:=t/l; a:=omega/4/d; b:=2×R; i:=1;
for alfa:=-s, -b×v, 0, b×v, s do
   begin if u<alfa then go to L; i:=i+1 end;
```

```
L : if i=3 ∨ i≧5 then ro:=4/3 × R↑3 × omega/d × v↑2;
G : go to A[i];
A1 :  F:=0;  go to out;  A2 :  F:=u+u↑2/2/s+s/2;
go to out;
A3 : gamma:= −b × v;  F:=s/2−ro;  go to M;
A4 : gamma:=b × v;  F:=s/2;
M :  F:=u−a/3/v × (u+gamma)↑3+F;  go to out;
A5 :  F:=u−u↑2/2/s+s/2−ro;  go to out;
A6 : F:=s−ro;  go to out;
out : end;
{folytatás}
```

Programunk globális változói *d*, *s*, *omega*, *R*, *t*, *l*, *u* és *F*. Utóbbi az eredmény tárolására szolgál. A program azon alapul, hogy a gép leszámlálja, *u* a szóba jövő egyenlőtlenségek közül hányadiknak tesz eleget, és az *A* kapcsoló segítségével definiált programágak közül annyiadikra tér rá. Az elágazást a *G* címkével ellátott vezérlésátadó utasítás biztosítja.

3. A 2. p.-ban szó volt arról, hogy **own** típusú változók esetén az őket tartalmazó blokkot úgy kell kiképezni, hogy a blokkba való első belépéskor a programnak más része működjék, mint az ismételt belépések alkalmával. A blokk megfelelő szervezését szintén egy kapcsoló segítségével biztosíthatjuk. A program sémája a következő lehet:

```
begin integer i;  {további deklarációk és utasítások};
i:=1;
   B : begin own real u;
      switch H:=H1, H2;                                    (4.21)
      go to H[i];
   H1: {utasítások, melyek előállítják az u változó kezdeti
      értékét};
      i:=2;  go to {folytatás};
```

*H2*: {a *B* blokk azon utasításai, melyek az első belépéskor nem működnek. Ezek az utasítások használják fel *u* megőrzött értékét};

{folytatás}: {a *B* blokk azon utasításai, melyek mindig működnek} **end**;

{utasítások}; **go to** *B*;

**end**

Az előzőkben több példát láttunk kapcsolókifejezésekre. A kapcsolókifejezés szintaktikus szempontból egy kapcsoló azonosítójából, és egy szögletes zárójelbe zárt aritmetikai kifejezésből áll. Utaltunk az indexes változókkal való formai és elvi analógiára is.

A kapcsolókifejezés speciális esete egy általánosabb szintaktikus kategóriának, melynek neve *helymegjelölő kifejezés*. A helymegjelölő kifejezések, akárcsak az aritmetikai és a logikai kifejezések, lehetnek feltétlenek és feltételesek. Foglalkozzunk először a feltétlen helymegjelölő kifejezésekkel. Definíciószerűen *feltétlen helymegjelölő kifejezésnek* nevezzük a címkéket és a kapcsolókifejezéseket együttesen. A *feltételes helymegjelölő kifejezés* általános alakja a következő:

**if** *L* **then** *H*1 **else** *H*2 (4.22)

ahol *L* logikai kifejezés, *H*2 tetszőleges, *H*1 pedig feltétlen helymegjelölő kifejezés. Miután *H*2 maga is lehet feltételes helymegjelölő kifejezés, hasonlóan az aritmetikai és logikai kifejezésekhez és a feltételes utasításokhoz, a szintaxis tetszőleges hosszúságú feltételláncok összeállítását megengedi. *H*1 csak feltétlen helymegjelölő kifejezés lehet, azonban (4.22)-re is érvényes az a szabály, amely szerint bármely feltételes helymegjelölő kifejezés zárójelbe zárva feltétlennek számít, és ily módon a **then** utáni **if** elkerülhető zárójelezés segítségével. A helymegjelölő kifejezések zárójelezésére aritmetikai (kerek) zárójeleket kell alkalmazni.

A most bevezetett új szintaktikus kategória segítségével megadjuk a **go to** utasítás és a kapcsolódeklarációk

szintaxisát és szemantikáját, most már teljes általánosságban. A **go to** utasítás általános alakja a következő:

$$\textbf{go to } H \tag{4.23}$$

ahol $H$ helymegjelölő kifejezés. Speciálisan, ha $H$ címke, akkor előttünk áll a vezérlésátadó utasításnak a IV. fejezet 3. p.-ban megismert alapesete. Ha $H$ kapcsolókifejezés, akkor a vezérlésátadó utasításnak a jelen pontban megismert általánosabb alakját kapjuk. [Lásd például (4.17).]

A kapcsolódeklarációk általános szintaxisa a következő:

$$\textbf{switch } S := D1, D2, \ldots, Dn \tag{4.24}$$

ahol $D1$, $D2$, ..., $Dn$ helymegjelölő kifejezések. A (4.16) kapcsolódeklaráció, melynek gyakorlati alkalmazásait a (4.19), (4.20) és (4.21) programokban láttuk, (4.24)-nek az a speciális esete, amikor a kapcsolólista valamennyi eleme címke.

Áttérünk most (4.23) és (4.24) szemantikájára. Először is, ha $H$ feltételes kifejezés [lásd (4.22)], akkor ki kell értékelni a benne szereplő $L$ logikai kifejezést, és ennek megfelelően kell választani a $H1$ és $H2$ kifejezések között. Ha a kiválasztott kifejezés maga is feltételes,* akkor a feltételek kiértékelését mindaddig kell folytatni, amíg feltétlen kifejezésre nem jutunk.

Ha a logikai feltételek kiértékelése során nyert feltétlen helymegjelölő kifejezés címke, akkor a vezérlés átadandó arra az utasításra, amely ezt a címkét viseli (röviden: erre a címkére); ha viszont kapcsolókifejezés, akkor a benne szereplő aritmetikai kifejezés kiértékelése után megkeresendő a megfelelő kapcsolólista annyiadik eleme, amennyi a kapott aktuális érték. Ha ez címke, akkor a vezérlés erre a címkére adandó át [lásd a (4.16) és (4.17)-tel kapcsolatban mondottakat], míg ha ismét kapcsolókifejezés, akkor a benne szereplő aritmetikai kifejezés kiértékelése után megkeresendő a megfelelő kapcsolólista megfelelő eleme, és ismét megvizsgálandó, hogy az címke vagy kap-

* $H1$ csak zárójelben lehet feltételes kifejezés.

csolókifejezés-e stb. Az eljárás addig folytatandó, amíg csak címkéhez nem jutunk, és a vezérlés a fent leírt eljárás végén erre a címkére adandó át.

Egy **go to** utasítás akkor értelmes, ha a fenti eljárás véges sok lépésben elvezet egy címkére, ellenkező esetben értelmetlen.

Beszélhetünk egy *helymegjelölő kifejezés „aktuális értékéről"*. Ez definíciószerűen az a címke, amelyre a tárgyalt algoritmus elvezet. E terminus alkalmazásával azt mondhatjuk, hogy egy vezérlésátadó utasítás értelmességéhez az szükséges, hogy a benne szereplő helymegjelölő kifejezés aktuális értékkel bírjon. Feltűnő az analógia az értékadó utasításokkal, amelyek akkor bírnak értelemmel, ha a jobb oldali kifejezésnek van aktuális értéke.

Példaként tekintsük a következő programsémát:

```
begin integer i, j;   switch A := A1, A2, B[i];
   switch B := if i = 1 then P else Q, B2;
   {utasítások};   i := 1;   j := 2;   {utasítások};          (4.25)
F : go to if i < j then A[i + j] else A2;   {utasítások}; end
```

Kövessük végig az *F* címkéjű vezérlésátadó utasítás végrehajtását, az elmondottak alapján! Először is, az adott aktuális értékek mellett, a logikai feltétel teljesül, ezért a két megadott helymegjelölő kifejezés közül a **then** utánit kell választanunk. Ez kapcsolókifejezés; a kapcsolóindex aktuális értéke 3. Meg kell tehát keresnünk az *A* kapcsolólistájának harmadik elemét. Ez *B*[*i*], ami ismét kapcsolókifejezés, a kapcsolóindex aktuális értéke 1. *B* kapcsolólistájának első eleme megint feltételes helymegjelölő kifejezés, és a benne szereplő logikai kifejezés aktuális értéke **true.** A **then** után most már a *P* címke áll, így végül a vizsgált **go to** utasítás a *P* címkére vezet, vagyis a helymegjelölő kifejezés aktuális értéke *P*.

Könnyű olyan helymegjelölő kifejezést mutatni, amelynek nincs aktuális értéke. Tekintsük például a következő programsémát:

```
begin integer i, j;
   switch A:=A1, A2, B[i];
   switch B:=B1, B2, A[j];                                    (4.26)
   {utasítások}; i:=j:=3; go to A[i];
{utasítások} end
```

A (4.26) programban olyan helymegjelölő kifejezés szerepel, amely a változók adott aktuális értéke mellett nem vezet címkére, hanem végtelen ciklust eredményez. A vezérlésátadó utasítás tehát nem hajtható végre.

## 5. A címkék hatásköre

A deklarált azonosítók hatáskörére vonatkozó szemantikai szabályokat az előzőekben — főként az 1. és 2. p.-ban — már megismertük. A címkék szintén lehetnek azonosítók, sőt leggyakrabban azok (lásd III. fejezet 3. p.), hatáskörükről mégis külön szemantikai szabályok rendelkeznek, minthogy nincs deklarációjuk.

Az alapvető szemantikai szabály a következő: minden címke lokális arra a legkisebb blokkra nézve, amelyik őt tartalmazza. Tekintsük például a következő programsémát:

```
begin {deklarációk}; {utasítások};
B : begin {deklarációk}; {utasítások};                        (4.27)
P : {utasítás}; {utasítások} end;
{utasítások} end
```

A (4.27) programséma két egymásba skatulyázott blokkból áll. A belső blokk, *B*, tartalmaz egy *P* címkével ellátott ALGOL-utasítást, tehát a *P* azonosító a *B* blokknak lokális azonosítója.

Ennek a szemantikai szabálynak néhány fontosabb következménye van. Először is, ha egy azonosítót, amely egy magasabb rendű blokkban deklarálva van (például egy skaláris változó azonosítója), címkeként használunk egy alacsonyabb rendű blokkban, ezzel de facto újradeklarál-

juk. tehát a reá való hivatkozás esetén érvénybe lépnek az újradeklarálásra vonatkozó szemantikai szabályok. Ha például a címkeként használt azonosító egy magasabb rendű blokkban skaláris változó azonosítója volt, akkor ennek a változónak az aktuális értéke (ha volt neki) azalatt, míg a gép a címkét tartalmazó legkisebb blokkot hajtja végre, látens állapotba kerül, és csak akkor válik ismét hozzáférhetővé, ha ebből a blokkból már kikerült a vezérlés.

A címkék lokalitási szabályának másik folyománya az a már ismert szemantikai szabály, amely megtiltja kívülről a blokk belsejébe való vezérlésátadást. Egy ilyen vezérlésátadás ui. nem lenne egyéb, mint egy blokk lokális azonosítójára való hivatkozás a blokkon kívülről. Tudjuk, hogy ez bármiféle lokális azonosító esetén a nyelv szemantikai szabályaiba ütközik. Összetett utasítás belsejébe szabad **go to** utasítással! beugrani.

Megtörténhet, hogy több hierarchiailag egymás fölé rendelt blokkban előfordul ugyanaz a címke. Ha egy **go to** utasítás erre a címkére hivatkozik, akkor a gép először megvizsgálja, hogy ugyanabban a blokkban előfordul-e ez a címke. Ha igen, a vezérlés erre a címkére adódik át. Ha a gép az azonos blokkban nem találja meg a rendeltetési címkét, megvizsgálja a következő legalacsonyabb hierarchiájú blokkot. Ha ebben megtalálja a keresett címkét, átadja rá a vezérlést, ha nem, ismét megvizsgálja a soron következő legalacsonyabb blokkot, s. í. t. A **go to** utasítás végrehajtása közben a vezérlés nem kerülhet alacsonyabb hierarchiájú blokkba, mint amilyenben a **go to** utasítás van.

Példaképpen tekintsük a következő programsémát:

```
A : begin {deklarációk};  {utasítások};
      P : {utasítás};
B :   begin {deklarációk};  {utasítások};                    (4.28)
        P : {utasítás};
C :     begin {deklarációk};  go to P;  {utasítások}
end end end
```

A (4.28) program három egymás fölé rendelt blokkból áll. A legalacsonyabb hierarchiájú *C* blokkban van egy vezér-

lésátadó **go to** utasítás, melynek rendeltetési címkéje *P*. Ebben a blokkban nem található meg a rendeltetési címke. A gép tehát keresni kezdi az eggyel magasabb hierarchiájú *B* blokkban, és meg is találja. A vezérlés erre a *B*-beli *P* címkére adódik át, bár az *A* blokkban is van egy *P* címke.

A **go to** utasítással kapcsolatban még egy szemantikai probléma vár tisztázásra. Tegyük fel, hogy valamely *A* blokk fejében deklarálva van egy *S* kapcsoló, melynek kapcsolólistáján szerepel egy sereg helymegjelölő kifejezés, ezek felépítésében pedig részt vesznek bizonyos azonosítók. Tegyük fel továbbá, hogy az *A* blokkba bele van skatulyázva egy *B* blokk. *B* fejében újradeklarálunk néhány *A*-beli azonosítót, köztük olyanokat is, amelyek *S* kapcsolólistáján szereplő helymegjelölő kifejezésekben előfordulnak. Mivel *B* része *A*-nak, *S* a *B* blokk globális azonosítója, tehát a *B* blokkban szabad az *S* kapcsolóra hivatkozni. Előfordulhat, hogy *S*-re hivatkozó kapcsolókifejezés aktuális értékének kiszámítása közben a kapcsolólistának egy olyan eleméhez jutunk el, amelynek egyes azonosítói a *B* blokk fejében újra lettek deklarálva. Kérdés, hogy ilyen esetben melyik deklaráció veendő figyelembe. Az ALGOL-jelentés a következőképpen rendelkezik erről az esetről: ha egy kapcsolólista elemeit felépítő azonosítót egy alacsonyabb rendű blokkban újradeklarálnak, akkor a kapcsolólista-elemek aktuális értékének kiszámítása közben úgy kell eljárni, mintha a belső blokkban nem újradeklarálás történt volna, hanem egy — a külső blokkbeli azonosítótól különböző — belső, lokális azonosító került volna deklarálásra; más szóval, szisztematikusan meg kell változtatni a belső blokk azonosítóit oly módon, hogy az újradeklarálás ténye megszűnjék. E szabály értelmében tehát az *A* blokkbeli, vagyis az újradeklarálás előtti azonosító veendő figyelembe.

Két példa talán megvilágítja a fentieket.

**1.** Tekintsük a következő programsémát:

```
A : begin integer i, j;
    switch S := if i < j then P else Q, R;
    i := 1;   j := 2;   {utasítások};                    (4.29)
```

*B* : **begin integer** *i*, *j*;
*i*:=2; *j*:=1; **go to** *S*[1];
{utasítások} **end**; {utasítások, melyek tartalmazzák a *P*, *Q* és *R* címkéket} **end**

Az előbbi szemantikai probléma a (4.29) programra vonatkoztatva azt a kérdést veti fel, hogy a **go to** *S*[1] utasítás végrehajtásakor kapcsolólista-elem aktuális értékének számításában a logikai feltétel **true** vagy **false** értékűnek veendő-e? Ha ti. az *i* és *j* változóknak *A* blokkbeli értékét vesszük figyelembe, akkor a logikai feltétel **true** értékű, ha pedig a belső *B* blokkbeli értékkel dolgozunk, akkor a logikai feltétel aktuális értéke **false**. A szabály szerint a külső blokkbeli értékek veendők figyelembe, ezért a logikai feltétel **true**-nak adódik.

Azt mondottuk, hogy a kétértelműség kiküszöbölése céljából a belső blokk azonosítóit oly módon kell (gondolatban) megváltoztatni, hogy az újradeklarálás ténye megszűnjék. Ez azt jelenti, hogy a *B* blokk átalakítandó a következő módon:

*B* : **begin integer** *ii*, *jj*;
*ii*:=2; *jj*:=1; **go to** *S*[1]; (4.30)
{utasítások} **end**

Most már nincsenek az *A* és *B* blokknak közös azonosítói, ezáltal megszűnt a kétértelműség.

Az azonosítók ilyen szisztematikus megváltoztatásának követelménye nem a programozóra, hanem a fordítóprogramra hárít kötelezettséget, tehát az átjelölést magának a gépnek kell automatikusan elvégeznie. A (4.30) program definíciós programként tekintendő, amelynek segítségével (4.29) szemantikáját definiáljuk. Hasonló definíciós programokkal találkoztunk a **for** utasítások tárgyalása során [lásd (3.50) és (3.53)].

**2.** A fentiek alapján világos a következő program jelentése is:

```
begin switch S:= P, Q, R;
   {utasítások};
   P : {utasítás};
   begin {deklarációk};  {utasítások};                    (4.31)
   P : {utasítás};  go to S[1];
{utasítás} end end
```

E példával kapcsolatban is felvethető a kérdés, hogy a belső blokkban levő kapcsolókifejezés aktuális értéke a két *P* címke közül melyik? Az előzőkben tárgyalt szabályok értelmében világos, hogy ez a címke a külső blokkbeli *P*, mivel a belső blokkban alkalmazott *P* címke szemantikai szempontból a *P* azonosító újradeklarálásának számít, és mint ilyenre, alkalmazni kell rá az azonosítók szisztematikus átjelölésének elvét, így úgy tekinthetjük, mintha azonosítója valami más, mondjuk *PP* lenne.

## 6. Magyarázó megjegyzések a programban

A deklarációk, mint az előző fejezetekben láttuk, lényegében a programra vonatkozó felvilágosítások, melyek elsősorban nem a programot felhasználó szakembernek, hanem a fordítóprogramnak szólnak, és lehetővé teszik, hogy a fordítóprogram előállítsa a célprogramot. A programban való eligazodás érdekében azonban nagyon hasznos, ha tartalmaz olyan felvilágosításokat is, amelyek a fordítóprogram számára ugyan értéktelenek, viszont tájékoztatják a felhasználót a program működéséről, alkalmazásának módjáról, a felhasználás esetleges korlátairól és a programmal kapcsolatos egyéb tudnivalókról. Az ALGOL nyelv lehetőséget ad arra, hogy a programozó ezeket a felvilágosításokat szintaktikusan korrekt módon beépítse a programba, mégpedig oly módon, hogy a fordítóprogram figyelmen kívül hagyja ezeket.

A magyarázó szövegek kezdetének jelzésére külön alapjel van bevezetve az ALGOL-ban, a **comment**,* amely

* **comment** = magyarázat, kommentár

**begin** és pontosvessző után alkalmazható. A magyarázatoknak a fordítóprogram által történő figyelmen kívül hagyását az a szemantikai szabály biztosítja, hogy a **comment** alapjeltől a legközelebbi pontosvesszőig minden kommentárnak számít.

Az **end** után **comment** nélkül is kezdhetők kommentárok: minden, ami az **end** alapjeltől a legközelebbi pontosvesszőig, **end**-ig vagy **else**-ig terjed, ugyancsak kommentárnak számít.

Példaként egészítsük ki a III. fejezet 8. p.-ban tárgyalt, lineáris egyenletrendszert megoldó (3.78) programot teljes ALGOL-programmá, és lássuk el tájékoztató megjegyzésekkel:

```
begin comment lineáris algebrai egyenletrendszer megoldása
Gauss—Jordan eliminációs módszer segítségével;
integer n;  {n beolvasása};
begin integer i, j, k;  real m;  array a[1:n, 1:n+1];
   {az együtthatómátrix beolvasása};
   comment Itt kezdődik a (3.78) program;
   for i:=1 step 1 until n do begin
   m:=1/a[i, i];
   for k:=i+1 step 1 until n+1 do a[i, k]:=m×a[i, k];
   for j:=1 step 1 until n do if j≠i then begin
      T:=a[j, i];  for k:=i+1 step 1 until n+1 do
      a[j, k]:=a[j, k]-T×a[i, k] end end i;
   {a megoldások kinyomtatása}
end end lineáris egyenletrendszer megoldása
```

Többszörösen egymásba skatulyázott blokkok, illetve összetett utasítások esetén gyakran nehéz áttekinteni, hogy melyik **begin**-hez melyik **end** tartozik. Alkalmas **end** utáni kommentárok segítségével nagymértékben fokozhatjuk a program áttekinthetőségét. Példánkban az *i* szerinti ciklust lezáró **end** utáni *i* betű, mint egybetűs kommentár, a ciklus végének jelzésére szolgál. Az **end**-ek utáni kommentárok annak ellenőrzésében is segítségünkre lehetnek. hogy minden megkezdett utasítás-zárójel be van-e zárva.

E helyen felhívjuk a figyelmet egy eléggé gyakori programozási hibára. A programozók sokszor elfeledkeznek az **end** utáni pontosvesszőről. Ez azzal a következménnyel jár, hogy az **end** után következő utasítást a fordítóprogram a legközelebbi **end**-ig, pontosvesszőig vagy **else**-ig kommentárnak tekinti, és nem fordítja le. Miután ily módon általában egy teljes utasítás marad ki, a megmaradó program szintaktikusan nem válik hibássá, és ezért a célprogram fog nemkívánt módon lefutni.

A kommentárokkal kapcsolatban ismertetett elhatárolási szabályoknak magától értetődő következménye, hogy a **comment** alapjellel bevezetett kommentár nem tartalmazhatja a pontosvesszó alapjelet, az **end** utáni kommentár pedig nem tartalmazhatja a pontosvesszó, az **end** és az **else** alapjelek egyikét sem.

## 7. Gyakorlatok.

1. Írjunk blokkfejet egy blokkhoz, melynek lokális azonosítói a következők:

Valós tipusú változók: *x*, *y*, *u*, *v*. Egész tipusú változók: *i*, *k*, *n*. Logikai tipusú változók: *q*, *ψ*. Tartalmaz ezenkívül a blokk két valós tömböt, *A*-t és *B*-t, melyek kétindexesek, az első indexek 0-tól 6-ig, a második indexek 1-től 20-ig változhatnak.

2. Egy ALGOL-program négy egymásba skatulyázott blokkból áll, az alábbi blokkfejekkel. A blokkfejek alapján állapitsuk meg az egyes változók tipusát és hatáskörét!

*B1* : **begin real** *A*, *B*, *J*, *K*; **integer** *SC*, *J6*, *i*, *p*; **Boolean** *gamma*;
{utasítások};

*B2* : **begin real** *A*, **Boolean** *J6*, *B*, *alfa*; **integer** *j*, *k*;
{utasítások};

*B3* : **begin real** *alfa*, *beta*; **integer** *m*, *n*; **Boolean** *gamma*;
{utasítások};

*B4* : **begin integer** *p*, *q*, *r*;
{utasítások}; **end end end end**

Az egyes azonosítókról tett megállapításainkat foglaljuk táblázatba a következő módon:

| | *B*1 | *B*2 | *B*3 | *B*4 |
|---|---|---|---|---|
| *A* | valós lokális | valós lokális | valós globális | valós globális |
| *B* | valós lokális | logikai lokális | logikai globális | logikai globális |
| *alfa* | nincs deklarálva | logikai lokális | valós lokális | valós globális |

Töltsük ki ezt a táblázatot valamennyi szereplő azonosítóra!

3. Mi a hiba a következő blokkfejben?

```
begin Boolean alfa, beta, gamma; real x, y;
  integer n, m; own array A [1:n, 1:m];
```

4. Egészítsük ki teljes programmá a III. fejezet 9. p.-jának 4. feladatát! A program operatív részeként felhasználhatjuk a könyv végén közölt megoldást.

5. Írjunk teljes ALGOL-programot az

$$F(x) = \int_0^x \frac{dt}{\sqrt{1 + 2t + \sin^2 t}}$$

határozatlan integrál tabellázására, 0-tól 1/2-ig 0.0001 lépésközzel!

Valamely függvény primitív függvényét (határozatlan integrálját) numerikusan az integrációs formulának „lépésenkénti" alkalmazásával számíthatjuk ki. Ez azt jelenti, hogy ha már ki van számítva a primitív függvény valamely $x$ helyen, akkor az $x + \Delta x$ helyen (Simpson-formulával való számolás esetén) az alábbi képlet szerint számíthatjuk ki:

$$F(x+\Delta x) = F(x) + \left[ f(x) + 4f\left(x + \frac{\Delta x}{2}\right) + f(x+\Delta x) \right] \frac{\Delta x}{3},$$

ahol $f(x)$ az integrál alatti függvényt jelenti.

Feladatunkat egy $x$ szerinti ciklussal oldhatjuk meg, amely 0-tól 1/2-ig megy. Esetünkben $\Delta x = 0.0001$. Jelezzük fiktiv (zárójelbe zárt) utasítással a primitív függvény értékének kinyomtatását minden egyes $x$ szerinti ciklusmenetben!

**6.** Ismeretes, hogy ciklus belsejébe nem szabad **go to** utasítással beugrani. Felmerülhet azonban a szükségessége olyan program írásának, amelyben (esetleg valamilyen logikai feltétel teljesülése esetén) **go to**-val ki kell lépni a ciklusból egy címkére, ott végrehajtani bizonyos utasításokat, majd ezek végeztével visszatérni a ciklusba, és ott folytatni azt, ahol végrehajtása abbamaradt. Készítsük el egy olyan ALGOL-program sémáját, amely a szintaktikus és szemantikai szabályokkal összeférő módon végrehajtja a fentieket!

**7.** Valamely ALGOL-programban definiálva van egy egész típusú *i* változó. Feltesszük, hogy *i* csak pozitív értékeket vesz fel. Vizsgáljuk *i* értékét modulo 3 és modulo 4. (Más szóval vizsgáljuk *i*-nek 3-mal és 4-gyel való osztási maradékát.) Ha *i* egyenlő *p*-vel modulo 3, és *q*-val modulo 4, akkor adjuk át a vezérlést az *Lpq* címkére. Például ha *i*-nek 3-ra vonatkozó osztási maradéka 2, és 4-re vonatkozó maradéka 1, akkor a vezérlés az *L*21 címkére adandó át. Írjuk fel egy olyan program sémáját, amely végrehajtatja a kívánt elágaztatást!

**8.** Írjunk teljes ALGOL-programot mátrixok szorzására. Az összeszorzandó mátrixok azonosítója legyen *A* és *B*. Az *A* mátrix legyen $n \times m$ típusú, a *B* mátrix pedig $m \times r$ típusú, mely esetben az eredmény $n \times r$ típusú. *n*, *m* és *r* maguk is változó adatok. Jelezzük fiktív utasításokkal az összeszorzandó mátrixok beolvastatását és az eredmény kiíratását!

**9.** Legyen adva egy háromelemű logikai vektor, melynek azonosítója *B*. *B* elemei egymástól függetlenül lehetnek **true** vagy **false** értékűek. Ily módon összesen 8 logikai kombináció jöhet létre. Legyenek *S*1, *S*2 és *S*3 a program különböző címkéi, és tegyük fel, hogy a program egy adott helyén vagy *S*1-re, vagy *S*2-re, vagy *S*3-ra kell átadni a vezérlést, mégpedig a következők szerint:

| | | | |
|---|---|---|---|
| ha | *B*[1] = **true**, | *B*[2] = **false**, | *B*[3] = **false** |
| vagy | *B*[1] = **true**, | *B*[2] = **true**, | *B*[3] = **false** |
| vagy | *B*[1] = **false**, | *B*[2] = **false**, | *B*[3] = **true** |

akkor *S*1-re,

| | | | |
|---|---|---|---|
| ha | *B*[1] = **false**, | *B*[2] = **true**, | *B*[3] = **false** |
| vagy | *B*[1] = **false**, | *B*[2] = **true**, | *B*[3] = **true** |
| vagy | *B*[1] = **true**, | *B*[2] = **true**, | *B*[3] = **true** |

akkor *S*2-re,

| | | | |
|---|---|---|---|
| ha | *B*[1] = **false**, | *B*[2] = **false**, | *B*[3] = **false** |
| vagy | *B*[1] = **true**, | *B*[2] = **false**, | *B*[3] = **true** |

akkor *S*3-ra.

Írjunk fel olyan vezérlésátadást, amely minél egyszerűbben elvezet a megfelelő címkére!

**10.** Tekintsük a következő programsémát:

```
A:begin real x; {utasítások};
  L:{utasítás};
B:begin L: {utasítás};
  {utasítások} end;
  go to L; {utasítások} end
```

A program végrehajtása során a két *L* címke közül melyikre adódik át a vezérlés?

Ahol szükséges, a megoldásprogramokat lássuk el tájékoztató kommentárokkal!

# Ötödik fejezet

# ELJÁRÁSOK

## 1. Bevezetés

A számológép-programozás gyakorlatában sokszor igen hasznos, ha egyes programokat olyan módon írnak meg, hogy azok bizonyos értelemben zárt egészet alkossanak. Egy ilyen program ui. építõkocka-elemként beleépíthetõ egy nagyobb programba — esetleg több helyen is —, ha szükséges. Különösen két esetben célszerû a programok egyes részleteit „elõregyártott elem" formájában kidolgozni:

1. ha egy nagyobb programban ugyanaz az utasításcsoport több különbözõ helyen jelenne meg;

2. ha a program valamely része bizonyos szempontból „közérdekû", azaz elõreláthatólag ugyanezt a programrészt a késõbbiekben más programok is fel tudják majd használni.

Az elsõ esetben azért célszerû a kérdéses részprogram építõkockaszerû kidolgozása, mert ezáltal elkerülhetõ azonos utasításcsoportok ismételt leírása, és ezzel a program térbeli terjedelme csökken. Az ilyen építõkockát, amelyet a továbbiakban *eljárásnak* fogunk nevezni, az ALGOL-ban mindig egy blokk fejében definiálják (deklarálják), és a blokk belsejében csak hivatkoznak erre a definícióra. A programozási folyóiratok a számítási eljárásokat (algoritmusokat) mindig ilyen ALGOL-eljárás formában közlik, és ezeket csak bele kell másolni a megfelelõ blokkba, ahol majd felhasználásra kerülnek.

Mi jellemzõ egy ilyen ALGOL-eljárásra?

Az ALGOL-eljárás nem egyéb, mint egy — megfelelõ kísérõ információval ellátott — ALGOL-program, amely meghatározott kiinduló információból meghatározott ki-

menő információt állít elő, mégpedig olyan módon, hogy a kiinduló információt mindig egy ALGOL-program adja meg az eljárás részére, és az eredmény további felhasználás céljából a gép rendelkezésére áll. Az ALGOL-eljárás sohasem teljes program, működtetéséhez mindig szükség van egy vezérprogramra, amely aktivizálja, azaz megadja részére a kiinduló információt, utasítást ad a végrehajtásra, és a végrehajtás után valamilyen módon felhasználja az eredményt. Az eljárás aktivizálását másképpen az *eljárás behívásának* nevezzük.

Eljárás lehet például egy olyan ALGOL-program, amely lineáris algebrai egyenletrendszereket old meg; kiinduló információ pedig az egyenletrendszer rendszáma, az együtthatómátrix és a jobb oldal; az eredmény a rendszer megoldása. Az előzőkben elmondottak szerint az együtthatómátrixot, a rendszámot és a jobb oldalt a vezérprogramnak kell „szállítania" az eljárás részére, az eljárás viszont „szállítja" a vezérprogram részére a megoldásokat, és ugyancsak a vezérprogramnak kell gondoskodnia ezek felhasználásáról. Ha a feladat olyan, hogy a számítás során többször kell lineáris egyenletrendszereket megoldani, akkor a vezérprogram többször is behívhatja ugyanazt az egyenletrendszer-megoldó eljárást.

Az eljárások alkalmazásának előnye az említett első esetben csupán a helytakarékosság, ami a célprogramban memóriatakarékossággal egyenértelmű. A második esetben azonban sokkal komolyabb jelentősége van: a programozónak nem kell tudnia, hogy az eljárás hogyan működik, elég, ha azt tudja, hogyan kell megadni a bemenő adatokat, és hogyan kapja meg az eredményt. (Természetesen ez csak akkor elegendő, ha a programozó kipróbált, hibátlan eljárást épít bele a programjába.) Előbbi példánknál maradva, egy lineáris egyenletrendszer-megoldó eljárás felhasználásához a programozónak nem kell tudnia, hogyan történik a Jordan-elimináció.

Az ALGOL-eljárások bevezetése rendkívüli módon megkönnyíti az irodalmilag publikált programok gyakorlati felhasználását, mert beépítésükhöz úgyszólván semmi előzetes átalakítást, átdolgozást, szerkesztési munkát stb.

nem kell végezni, ha a felhasználó gépnek megszorításoktól eléggé mentes ALGOL-fordítója van.

E helyen szeretnénk az Olvasó figyelmét felhívni azokra a folyóiratokra, amelyek rendszeresen publikálnak ALGOL-eljárásokat. Ilyenek a Communications of the ACM,* a BIT,** a Numerische Mathematik, a Journal of the ACM stb. Az elmúlt öt év alatt ezek a folyóiratok több száz algoritmust közöltek a legkülönbözőbb numerikus matematikai feladatokra, és a gyakrabban előforduló feladattípusokra csaknem mindig található ezek között megfelelő. Ennek folytán sok esetben egy feladat programozási munkája a megfelelő eljárás kiválasztására, a programba való bemásolására és alkalmas vezérprogram írására redukálódik.

A könyv hátralevő részében az eljárásokkal kapcsolatos szintaktikus és szemantikai szabályokat tárgyaljuk, továbbá példákkal illusztráljuk az ALGOL-eljárások megadási módját és gyakorlati alkalmazásukat.

## 2. Eljárásdeklaráció és eljárásutasítás. Aktuális és formális paraméterek

Említettük az első fejezetben, hogy az ALGOL-eljárásokat mindig egy blokk fejében definiálják, és az aktivizálás a blokk belsejében, az eljárásra való hivatkozással történik.

Minden eljárásnak azonosítója van, és a blokkfejben lényegében az azonosítót deklarálják. Az eljárásdeklaráció egy különleges ALGOL-program, amely az eljárás azonosítójához rendelt algoritmus leírását tartalmazza.

Tudjuk az előzőkből, hogy az eljárás részére mindig meg kell adnunk valamilyen kiinduló információt, és az eljárás ennek felhasználásával előállít valamilyen kimenő információt. Az eljárás az őt aktivizáló programmal bizonyos azonosítókon keresztül tartja a kapcsolatot: a bemenő in-

* ACM = Association for Computing Machinery.
** BIT = Nordisk Tidskrift for Informations Behandling.

formáció megadása lehet pl. értékadás bizonyos meghatározott változókra nézve, a kimenő információt pedig ugyancsak bizonyos meghatározott változóknak adja az eljárás. Azokat az azonosítókat, amelyeket arra a célra jelölünk ki, hogy rajtuk keresztül az eljárás kapcsolatot tartson az őt behívó programmal, az *eljárás paramétereinek* nevezzük.

Az eljárásdeklaráció a **procedure*** alapjellel kezdődik,** ezt az eljárás azonosítója követi. Az eljárásazonosító után, kerek zárójelbe zárva, az eljárás formális*** paramétereinek listája áll, melyet egymástól vesszővel elválasztott azonosítók sorozata alkot. Mindezt egy pontosvessző követi.

Példa eljárásdeklaráció kezdetére:

**procedure** *Gamma* (*x*, *y*, *z*); (5.1)

Itt *Gamma* az eljárás azonosítója, *x*, *y*, *z* pedig formális paraméterek. Megemlítjük, hogy a formális paraméterek listája bizonyos körülmények között hiányozhat, amire később példát is mutatunk majd.

Ezután az eljárásdeklaráció egy kísérő informatív részszel folytatódik, amely a fordítóprogram számára szükséges felvilágosításokat tartalmazza az eljárás formális paramétereire vonatkozólag. Bizonyos körülmények között ez a programrész is hiányozhat. Az eljárásdeklaráció e részét *specifikációs résznek* nevezzük.

A specifikációs részt követi az az ALGOL-program, amely leírja magát az algoritmust, amelyet az eljárás azonosítójához rendelünk. Ezt az ALGOL-programot az *eljárás törzsének* nevezzük. Az (5.1) típusú bevezető jelsorozat és a specifikációs rész együttesen alkotja az *eljárásfejet*. Az eljárásdeklaráció tehát eljárásfejből és eljárástörzsből áll. Ez a felépítés némileg hasonló a blokkok szerkezetéhez.

Példaként lássunk egy eljárásdeklarációt, amely abszolút értékkel és fázisszöggel megadott komplex számokat átalakít valós és képzetes részükkel ábrázolt számokká. A bemenő információ az abszolút érték és a fázisszög, a

* **procedure** = eljárás.
** Kivétel a függvényt definiáló speciális eljárás, l. 4. p.
*** A „formális" jelző értelmét a későbbiekben magyarázzuk meg.

kimenő információ pedig a valós és a képzetes rész. Egyszerűség kedvéért feltesszük, hogy a specifikációs rész hiányzik a programból. Ekkor az eljárásdeklaráció a következő alakot öltheti:

**procedure** *ÁTALAKÍTÁS* (*x*, *y*, *ro*, *fi*);
  **begin** *x*:=*ro* × *cos* (*fi*);  (5.2)
  *y*:=*ro* × *sin* (*fi*) **end**;

Itt a formális paraméterek *x*, *y*, *ro* és *fi*. Ezek azok az azonosítók, amelyeken keresztül az eljárás kapcsolatot tart azzal a programmal, amely aktivizálni fogja. Emlékeztetünk rá, hogy az eljárásdeklaráció mindig egy blokk fejében van elhelyezve, tehát (5.2) valamely blokk deklarációs részének része. Az eljárástörzs értelméből következik, hogy itt *ro* és *fi* „bemenő" paraméterek, tehát az eljárás részére megadandó bemenő információ hordozására szolgálnak, míg *x* és *y* „kimenő" paraméterek, vagyis az eredmény közvetítésére használhatók. A formális paraméterek listáján a be- és kimenő paraméterek semmilyen látható módon nincsenek megkülönböztetve, sorrendjük tetszőleges, elkülönítésükről semmiféle szabály nem rendelkezik. Egy paraméter lehet egyszerre bemenő és kimenő is, tehát ezt a megkülönböztetést, amelyet kezdetben a dolog lényegének megértetése céljából használtunk, a későbbiekben teljesen el fogjuk ejteni, és közös néven mindig csak „formális" paraméterről fogunk beszélni.

Egy ALGOL-eljárás törzse mindig egyetlen utasítás. Ha több utasításból áll, akkor ezeket utasítás-zárójelekkel egyetlen utasítássá kell összefogni. A gyakorlatban az eljárástörzs általában blokk; az ALGOL szintaktikus szabályai megengedik az eljárástörzset megnyitó **begin** után lokális azonosítók deklarálását.

Az eljárások az ún. *eljárásutasítások* segítségével aktivizálhatók. Az eljárásutasítás az aktivizálandó eljárás azonosítójából, és a bemenő információt közvetítő, ill. az eredmény átvételéről rendelkező aktuális paraméterek listájából áll. Például az (5.1) kezdetű eljárás egy lehetséges aktivizálása:

*Gamma* (*p*, *q*, *r*);  (5.3)

Az (5.2) eljárás egy lehetséges aktivizálása pedig:

*ÁTALAKÍTÁS* (*u*, *v*, *alfa*, *beta*); (5.4)

(5.3)-ban *p*, *q* és *r*, (5.4)-ben *u*, *v*, *alfa* és *beta* aktuális paraméterek. Az aktuális paraméterek száma mindig megegyezik a formális paraméterek számával. A *Gamma* eljárásnak például három formális paramétere van az eljárásdeklarációban, ezért minden rá való hivatkozásnál három aktuális paramétert kell megadni. Az *ÁTALAKÍTÁS* nevű eljárás paramétereinek száma négy.

(5.3) és (5.4) segítségével az eljárásutasítások szintaxisát illusztráltuk. Most áttérünk a szemantikai tárgyalásra, azaz annak ismertetésére, mi játszódik le az eljárások aktivizálásakor.

A részleteket egyelőre mellőzve, egy eljárásutasítás funkciója a következő:

1. Közvetíti az eljárás részére a kiinduló információt;
2. végrehajtatja az eljárástörzset;
3. gondoskodik arról, hogy az eredmények a megfelelő helyre kerüljenek;
4. az eljárástörzs végrehajtása után vezérlésátadást hajtat végre az eljárást behívó program folytatására; ilyenkor általában - de nem okvetlenül — az eljárásutasítást a felírás sorrendjében követő utasítás kerül végrehajtásra.

A továbbiakban annak részletes tárgyalásával foglalkozunk, hogyan kell végrehajtani egy eljárásutasítást. Tekintsük például az (5.2) eljárásdeklarációt, és az (5.4) eljárásutasítást.

Ahhoz, hogy (5.2) és (5.4) szintaktikusan és szemantikusan korrekt módon előfordulhassanak egy programban, bizonyos feltételeknek kell teljesülniök. Először is, az *ÁTALAKÍTÁS* eljárásnak egy olyan blokk fejében kell deklarálva lennie, amelyben *u*, *v*, *alfa* és *beta* lokális vagy globális változók. Másodszor, az aktivizálást megelőzően *alfa* és *beta* valamilyen módon értéket kell, hogy kapjanak. E két feltételnek megfelelően, a következő vázlatos programszerkezetet kell megkövetelnünk:

```
begin real u, v, alfa, beta;
procedure ÁTALAKÍTÁS (x, y, ro, fi);
   begin x:=ro × cos (fi);
      y:=ro × sin (fi) end;                                    (5.5)
{utasítások, amelyek értéket adnak alfának és betának};
ÁTALAKÍTÁS (u, v, alfa, beta);
{Utasitások, amelyek felhasználják u-t és v-t} end
```

Mint mondottuk, *u*, *v*, *alfa* és *beta* lehetnek a blokk globális azonosítói is, tehát deklarációjuk lehet egy magasabb rendű blokkban.

Az (5.5) programban található eljárásutasítás hatására annak a komplex számnak, amelynek abszolút értéke *alfa* aktuális értéke, és fázisszöge *beta* aktuális értéke, valós része átadódik az *u*, képzetes része átadódik a *v* változónak. A megfelelő értékek elhelyezése után a program végrehajtása az eljárásutasítást követő utasítással folytatódik. A programozó számára tehát a program teljesen úgy működik, mintha az eljárásutasítás helyén a következő utasítások állnának:

```
u:=alfa × cos (beta);
v:=alfa × sin (beta);                                          (5.6)
```

Valóban, éppen ezzel definiáljuk az eljárásutasítás hatását: az eljárásutasítás lefordítása után olyan program jön létre, amely úgy működik, mintha az eljárásutasítás helyére bemásoltuk volna az eljárástörzset úgy, hogy előzőleg az eljárástörzsben szereplő formális paramétereket kicseréltük az eljárásutasításban szereplő aktuális paraméterekkel.

Valójában a fordítóprogram nem feltétlenül másolja bele az eljárástörzset a célprogramnak minden olyan helyére, ahol az eljárásra hivatkozás történik, hanem esetleg vezérlésátadásokat generál az eljárástörzs kezdetére és a program folytatására.

Az előzőkben elmondottak megmagyarázzák az eljárásdeklaráció paramétereire alkalmazott „formális" jelzőt. E jelző azt kívánja érzékeltetni, hogy aktivizáláskor

a gép nem a formális, hanem a helyükre lépő aktuális paraméterekkel hajtja végre az eljárást. Az aktuális paraméterek minden egyes behíváskor mások lehetnek. Például az (5.5) program (5.4)-en kívül tartalmazhatja az (5.2) eljárás egy másik aktivizálását is, mondjuk *w*, *z*, *pszi* és *khi* aktuális paraméterekkel, ha e paraméterek is deklarált változói az (5.5) blokknak, vagy egy magasabb rendű blokknak, és *w* és *z* az eljárás behívását megelőzően valami módon értéket kapott. A megfelelő eljárásutasítás a következő lehet:

*ÁTALAKÍTÁS* (*w*, *z*, *pszi*, *khi*); (5.7)

Az (5.7) eljárásutasítás végrehajtása az (5.6)-hoz teljesen hasonló módon átalakított értékadó utasítások végrehajtását jelenti, de az *u*, *v*, *alfa* és *beta* változók helyett ezúttal a *w*, *z*, *pszi*, illetve *khi* változókkal.

Magát az eljárásdeklarációt úgy kell képzelnünk, mint egy kitöltetlen kérdőívet, melynek kitöltésre váró rovatai a formális paraméterek. Az (5.2) eljárásdeklarációt a következő „kérdőívvel" asszociálhatjuk:

```
procedure ÁTALAKÍTÁS (....., ....., ....., .....)
                      1. par. 2. par. 3. par. 4. par.
begin .....:= ..... × cos(.....);
          1. par.  3. par.  4. par.
      .....:= ..... × sin(.....);                    (5.8)
          2. par.  3. par.  4. par.
   go to ..................................
                (hová kell visszatérni?)
end
```

Az eljárás aktivizálásakor valójában ezt a „kérdőívet" töltjük ki, majd pedig végrehajtjuk a „kitöltött" eljárástörzset. A kitöltéskor az eljárástörzs minden olyan helyére, amely alá az „1. par." szöveg van írva, beírjuk az aktuális paraméterlistán első helyen álló aktuális paramétert, és így tovább. A „hová kell visszatérni" rovatba pedig a következő ALGOL-utasítás címkéje kerül (ha nincs neki, akkor a fordítóprogram automatikusan elhelyez egy fiktív cím-

két). Ha a program működése közben az eljárás újabb aktivizálására kerül sor, a kitöltés újból megtörténik, az újabb aktuális paraméterekkel.

Ennek a kitöltési sémának az ismeretében kézenfekvően adódik egy sereg szintaktikus és szemantikai szabály az eljárások aktivizálásával kapcsolatosan.

Könnyű észrevenni többek között, hogy az eljárástörzs kitöltése után korrekt ALGOL-programot kapunk akkor is, ha a paraméterlista harmadik és negyedik helyére nem változónevet, hanem aritmetikai kifejezést írunk. Ha ellenben az első vagy második paramétert helyettesítjük aritmetikai kifejezéssel, akkor a szintaxissal ellenkező jelsorozatra jutunk, mert akkor egy értékadó utasítás bal oldalára kerül aritmetikai kifejezés. Például ha egy eljárásutasítás az (5.2) eljárást a következőképpen hívja be:

$$\textit{ÁTALAKÍTÁS}\ (w,\ z,\ 1 + pszi\uparrow 2,\ m\tilde{u} + n\bar{u} + khi); \qquad (5.9)$$

akkor az (5.8) program kitöltése után az alábbi eljárástörzs adódik:

$$\begin{aligned} &w := (1 + pszi\uparrow 2) \times cos\,(m\tilde{u} + n\ddot{u} + khi);\\ &z := (1 + pszi\uparrow 2) \times sin\,(m\tilde{u} + n\bar{u} + khi);\\ &\textbf{go to}\ \{\text{folytatás}\}; \end{aligned} \qquad (5.10)$$

Levonhatjuk tehát azt a szabályt, hogy ha egy formális paraméter nem szerepel az eljárástörzsben értékadó utasítás bal oldalán, akkor a neki megfelelő aktuális paraméter lehet aritmetikai kifejezés is. Ezt külön szabályként nem szokás kimondani; ennél általánosabb az ALGOL ama rendelkezése, hogy minden olyan helyettesítés megengedett, amelynek folytán az eljárástörzs „kitöltése" [vö. (5.8)] után szintaktikusan és szemantikusan helyes ALGOL-program keletkezik. Ebben speciális esetként benne van az imént kimondott szabály is.

Kézenfekvő, de talán nem árt megemlíteni, hogy ha egy eljárás aktivizálásakor valamelyik aktuális paraméter nem változó, hanem kifejezés, akkor ez elsődleges kifejezésként, tehát szükség esetén zárójelbe zárva helyettesítendő

be az eljárástörzsbe. E kitétel értelmét az alábbi példán illusztrálhatjuk:

```
begin real a, b, c;
procedure ILLUSZTR (A, B, C);
    C:= A/B;
comment Az eljárástörzs egyetlen utasításból áll, ezért az
utasítászárójelek szükségtelenek. A következő utasítás már
a blokktörzshöz tartozik;
    a:= 1;   b:= 5;   ILLUSZTR (3, a+b, c)
end
```

Ha az *ILLUSZTR* eljárás aktivizálásánál a fenti szabályt nem vesszük figyelembe, akkor az *A* formális paramétert 3-mal, a *B* formális paramétert pedig $a+b$-vel helyettesítve a következő helytelenül átalakított eljárástörzset kapjuk:

$$c:=3/a+b$$

Szabályunk alapján ezzel szemben a kifejezésparamétert szükség esetén zárójelben kell alkalmazni, ezért a helyesen átalakított eljárástörzs a következő:

$$c:=3/(a+b)$$

Az (5.10) programra visszatérve, annak szemlélése közben nyilván feltűnik az Olvasónak, hogy abban — fölöslegesen — kétszeres számolás történik: az aktuális paraméterként megadott aritmetikai kifejezések kétszer kerülnek kiszámításra. Valóban, ha egy formális paramétert kifejezésparaméterrel helyettesítünk, akkor ez a kifejezés az eljárástörzs minden olyan helyén kiszámításra kerül, ahol a megfelelő formális paraméterre hivatkozás történik. Ez annak a mechanizmusnak a következménye, amely az aktuális paramétereket behelyettesíti a formális paraméterek helyébe.

Kézenfekvő tehát az a kívánalom, hogy jó lenne valami módon a fölösleges ismétlésektől megszabadulni. Az lenne kellemes, ha a gép — ahelyett, hogy minden alkalommal újra meg újra kiszámítja —, inkább egyszer kiszámítaná

előre a paraméterkifejezést, tárolná az eredményt, és többszöri hivatkozás esetén egyszerűen elővenné egy segédváltozóból (munkarekeszből). Ha a programozó az (5.10) programot maga írná fel, így oldaná ezt meg:

```
m1 := 1 + pszi ↑2;   m2 := mũ + nü + khi;
w := m1 × cos (m2);  z := m2 × sin (m2);          (5.11)
go to {folytatás};
```

Azt kívánjuk tehát, hogy a gép maga is képes legyen ilyenformán átírni az eljárástörzset.

Az első megoldás az lehet, hogy az (5.9) eljárásutasítás elé beiktatunk két értékadó utasítást:

```
m1 := 1 + pszi ↑2;   m2 := mü + nü + khi;
ÁTALAKÍTÁS (w, z, m1, m2);                        (5.12)
```

és ennek hatására már pontosan az történik, amit (5.11)-gyel kapcsolatban kívántunk. Azonban a programozók szeretnek minél kevesebbet írni, és ezért azt szeretnék, hogy ezeket az értékadó utasításokat maga a gép írja fel.

Ezzel elérkeztünk az ALGOL egyik igen fontos fogalmához, az ún. *érték szerinti paraméterhíváshoz.*

Világos, hogy az aktuális-formális paramétercserének az a módja, ahogy azt (5.8)-cal kapcsolatban leírtuk, szükséges és bizonyos esetekben nem helyettesíthető mással. A nyelv azonban a paramétercsere egy más módját is lehetővé teszi. A csere tehát kétféleképpen valósulhat meg: név szerint [ez felel meg az (5.8)-cal kapcsolatban mondottaknak] és érték szerint [ez az (5.12)-vel kapcsolatban mondottak ALGOL-nyelvi megfelelője].

A programozónak lehetősége van megjelölni azokat a formális paramétereket, amelyekre az (5.12) típusú kezelést igényli, tehát amelyekre az eljárástörzsbe való behelyettesítés helyett előzetes kiértékelést kíván. Az ilyen érték szerint hívott paraméterek megjelölése a **value*** alapjel segítségével történik: ezeket a paramétereket a formális paraméterek listája után, egy **value** kezdetű listán

* **value** = érték.

fel kell sorolni. Ha például az (5.2) eljárásdeklaráció *ro* és *fi* paramétereire az érték szerinti behívást kívánjuk alkalmazni, akkor az eljárásdeklarációt így kell kezdenünk:

**procedure** *ÁTALAKÍTÁS* (*x*, *y*, *ro*, *fi*); **value** *ro*, *fi*;

Az érték szerint hívott paraméterek típusát mindig meg kell adni egy a **value**-listát követő újabb listán. Esetünkben *ro* és *fi* valós típusúak, és típusuk megadása teljesen úgy történik, mintha egy blokkfejben deklarálnánk őket:

**real** *ro*, *fi*;

Ez azonban nem deklaráció, hanem specifikáció.

Ha az eljárásnak vannak más jellegű — például egész típusú — érték szerint hívott formális paraméterei, akkor ezeket egy **integer** kezdetű listán soroljuk fel. A **value**-listát az őt követő specifikációkkal együtt az *eljárásdeklaráció specifikációs részének* nevezzük. A fent megadott módon specifikálni kell minden érték szerint hívott formális paramétert, de szabad bármely formális paramétert is (lásd a 3. p.-ot).

Azon formális paraméterek esetében, amelyek az eljárásdeklarációban, az érték szerint hívott paraméterek listáján* fel vannak sorolva, az eljárás aktivizálásába definíciószerűen beleértendő azoknak az értékadó utasításoknak a végrehajtása, amelyek bal oldalán a formális, jobb oldalán pedig a neki megfelelő aktuális paraméter áll. Ezek az értékadó utasítások nincsenek explicite jelen a programban, elhelyezésükről a fordítóprogram gondoskodik. Fontos megjegyeznünk, hogy ezeknek a fiktív értékadó utasításoknak a végrehajtására az eljárástörzs végrehajtása *előtt* kerül sor.

Azoknak a formális paramétereknek aktuális paraméterekkel való kicserélése, amelyek nem szerepelnek az érték szerint hívott paraméterek listáján, a korábban leírt módon, az (5.8)-cal kapcsolatban elmondottak szerint történik.

* E listát zsargonban value-listának szokás nevezni.

Ha például az (5.2) eljárásdeklarációt a következőképpen adjuk meg:

**procedure** *ÁTALAKÍTÁS* (*x*, *y*, *ro*, *fi*); **value** *ro*, *fi*;
**real** *ro*, *fi*;
**begin** *x*:=*ro* × *cos* (*fi*); *y*:=*ro* × *sin* (*fi*); **end**;

és ezt az eljárást az (5.9) eljárásutasítással aktivizáljuk, a gép a következő programot fogja végrehajtani:

*ro*:=1+*pszi*↑2; *fi*:=*mü*+*nü*+*khi*;
*w*:=*ro* × *cos* (*fi*); *z*:=*ro* × *sin* (*fi*); **go to** {folytatás}; (5.13)

Az érték szerint hívott formális paraméter tehát nem „kitöltetlen rovat", mint a név szerint hívott, hanem munkarekesz, segédváltozó; szerepe teljesen hasonló az (5.12) programban bevezetett *m*1, ill. *m*2-höz. Innen kapta ez a fajta paraméterbehívás a nevét; ebbe a munkarekeszbe az eljárás aktivizálását megelőzően beíródik az aktuális paraméter aktuális értéke, és az eljárás végrehajtása során a gép ezzel az értékkel dolgozik.

Az (5.10) programmal kapcsolatban felvetett problémát tehát megoldattuk, lehetőségünk van az ALGOL-ban kikötni az előzetes értékadást. A paraméter behívásának típusát azonban az eljárásdeklaráció írásakor egyszer s mindenkorra el kell döntenünk, mivel az érték szerint hívott paraméterek listája a program egész menetére nézve megszabja a paraméter név, illetve érték szerinti kezelését.

Gyakorlati alkalmazásokra a következő pontban látunk majd példákat. Azonban már e helyen megjegyezzük, hogy a kétféle paraméterhívás között mindig az adott program szükségletei szerint kell választanunk. Látni fogjuk, hogy lehetnek egy eljárásnak olyan paraméterei is, amelyek eleve nem hívhatók érték szerint. (Ilyen például az *ÁTALAKÍTÁS* eljárásban *x* és *y*.)

Befejezésül, esetleges félreértések elkerülése érdekében még egyszer hangsúlyozzuk, hogy a fentiekben lényegében állandóan arról volt szó, hogyan fordítja le a fordítóprogram az eljárásutasításokat, és milyen átalakításokat hajt végre az eljárástörzsön. Tehát például az érték szerint hí-

vott formális paraméterek esetében, bár a kényelmes beszédmód érdekében néha azt mondjuk, hogy „értékadó utasításokat hajtunk végre", ezt mindig úgy kell érteni, hogy a fordítóprogram értékadó utasításokat generál és helyez el a célprogramban, amelyek az aktuális paraméterek aktuális értékeit átadják a formális paramétereknek.

## 3. Példák eljárások alkalmazására. A specifikációs rész

Ebben a fejezetben az előzők jobb megvilágítását célzó gyakorlati példákat tárgyalunk.

Első példaként egy olyan eljárást vizsgálunk, amely másodfokú algebrai egyenleteket old meg, a megoldóképlet segítségével. Az eljárástörzs rendelkezésünkre áll: a III. fejezet 5. p.-ban már vizsgáltuk a másodfokú egyenlet megoldásának programját, mint kiragadott programrészletet, kísérő információ nélkül. Most ezt a programot [lásd (3.40) alatt] kiegészítjük eljárásdeklarációvá. Az eljárásdeklaráció a következő lehet:

```
procedure MEP (a, b, c, x1r, x2r, x2i, B, EXIT);
value a, b, c;   real a, b, c, x1r, x1i, x2r, x2i;
Boolean B;   label EXIT;
begin real M, P;                                                    (5.14)
   if a=0 ∧ b=0 then go to EXIT;
   if a≠0 then begin M:=b↑2−4×a×c;   P:=sqrt(abs(M));
      if M≧0 then begin x1i:=x2i:=0;
         x1r:=(−b+P)/2/a;   x2r:=(−b−P)/2/a;
         if abs (x1r)>abs (x2r) then x2r:=c/x1r else x1r:=
         c/x2r
      end else begin
   x1r:=x2r:=−b/2/a;   x1i:=P/2/a;   x2i:=−x1i   end;
B:=true end else begin
x1r:=x2r:=−c/b;
x1i:=x2i:=0;   B:=false end end MEP;
```

A *MEP* eljárás az *a*, *b*, *c* együtthatókból kiszámitja a másodfokú egyenlet *x1r*, *x1i*, *x2r*, *x2i* megoldásait, valamint előállít egy logikai értéket, amely **true** vagy **false** aszerint, hogy az egyenlet valódi másodfokú-e, vagy elfajult. Magának az eljárástörzsnek az elemzésére nem térünk vissza, szólnunk kell ellenben a formális paraméterekről.

Az eljárás formális paraméterei nyilván azok az azonositók, amelyek szükségesek az eljárás, és az őt behívó program kapcsolatának fenntartásához. Ilyenek mindenekelőtt a három bemenő paraméter (*a*, *b* és *c*), valamint az öt kimenő paraméter (*x1r*, *x1i*, *x2r*, *xri* és *B*). A bemenő paramétereket érték szerint hivottnak választottuk abból a meggondolásból, hogy név szerinti hivással igen sok fölösleges számolást végezne a gép olyan esetekben, ha az aktuális paraméterek aritmetikai kifejezések. (Gondoljuk meg, hány helyre kellene az aktuális paraméterkifejezéseket bemásolni az eljárástörzsbe!)

A kimenő paraméterek szükségképpen név szerint hivottak. Ezt megértjük, ha meggondoljuk, hogy az érték szerinti hivás értékadást jelent a formális paraméterre, az eljárástörzs végrehajtását megelőzően. Ha a tárgyalt sémát az eljárás egy kimenő paraméterére alkalmazzuk, értelmetlenségre jutunk, mert ez az értékadás az aktuális paraméterváltozó eljárás végrehajtása előtti értékét adná a formális paraméterváltozónak, ezzel szemben a számitás eredményének közlése a behivóprogrammal nem volna biztositva. Ha az eljárás aktivizálását megelőzően az aktuális paraméternek nem volt értéke – ez igen könnyen előfordulhat, hiszen éppen arra szolgál az eljárás, hogy értéket adjon neki –, akkor az eljárás behivásakor olyan értékadó utasitást próbálna a gép végrehajtani, amelynek jobb oldala nincs definiálva. Jól jegyezzük meg tehát: bemenő paraméterekkel kapcsolatban helye van megfontolásoknak az érték, vagy a név szerinti hivás választását illetően, *kimenő paraméterek esetében* viszont *csak a név szerinti hivás használható*.

Van az eljárásnak egy nyolcadik paramétere, amelyről még nem szóltunk. Ez jellegében különbözik az előzőktől, amennyiben nem változó, hanem címke. A címkeparamété-

reket, amelyek csak név szerint hívhatók,* ugyanúgy kell kezelni, mint a formális változókat, tehát az (5.8)-cal kapcsolatos behelyettesítési sémát kell rájuk alkalmazni. Ez annyit jelent, hogy az eljárástörzsben a **go to** *EXIT* utasítás éppen olyan „kitöltetlen rovat", mint amilyenek a név szerint hívott formális paraméterekkel kapcsolatban keletkeznek, és az eljárás aktivizálásakor a fordítóprogram „tölti ki" ezt a rovatot az aktuális paraméterként megadott címkével. Ha pl. az aktuális címkeparaméter *STOP*, akkor a célprogram a **go to** *STOP* utasítást fogja végrehajtani, ha az eljárástörzs végrehajtása közben azt tapasztalja, hogy az $a$ és $b$ együtthatók mindketten nullák. A címkeparaméterek specifikálására az ALGOL-ban külön alapjelet vezettek be, ez a **label**.** A *MEP* eljárásban minden formális paramétert specifikáltunk, bár elég lett volna az érték szerint hívottakat specifikálni. Ezzel kapcsolatban megemlítjük, hogy a gépi reprezentánsok általában előírják valamennyi formális paraméter specifikálását, mert ez egyszerűsíti a fordítóprogramot.

Az (5.14) eljárásdeklaráció önmagában nem program, csak valamely blokk fejének kiragadott részlete. Eljárásunk gyakorlati alkalmazásaként írjunk programot a következő transzcendens egyenletrendszer megoldására:

$$\begin{aligned} px^2 + \sqrt{1+p}\ x - 1.1046 &= 0 \\ (x^2+1)p^2 + \sqrt{1-x^2}\,p - 0.73 &= 0. \end{aligned} \tag{5.15}$$

(5.15) megoldására a következő ad hoc iterációs algoritmust alkalmazhatjuk: kiindulva valamely $p_0$ kezdőértékből, a rendszer első egyenletét (amely másodfokú $x$-ben) megoldjuk $x$-re. Az így kapott $x$ értékkel megoldjuk a második egyenletet $p$-re, (ez $p$-ben másodfokú). Ily módon $p$-nek egy javított értékéhez jutunk, ezzel ismét az első egyenletet oldjuk meg $x$-re, majd a másodikat $p$-re, és így tovább. Az iterációt akkor hagyhatjuk abba, ha mind $p$-nek, mind

* A hivatkozási nyelv nem, de a gépi reprezentánsok általában kizárják címkék érték szerinti hívását, ezért ezzel nem foglalkozunk e könyv keretében.

** **label** = címke.

$x$-nek két egymás után következő iteráltja kevesebbel tér el, mint $10^{-6}$. A $p_0$ kezdőérték választható 0.55-nek, ebben az esetben a leírt iteráció konvergálni fog (5.15)-nek egy megoldásrendszeréhez. Teljes program írandó, az eredmények kinyomtatását végző utasításokat szimbolizáljuk a szokott módon!* Az iteráció során a megoldóképletben a négyzetgyököt pozitív előjellel vesszük tekintetbe, azaz *MEP* kimenő paraméterei közül $x1r$-et használjuk fel. Feladatunkat a következő program segítségével oldjuk meg:

**begin real** $x$, $xE$, $p$, $pE$, *DUMMY1*, *DUMMY2*,
*DUMMY3*; **integer** $i$; **Boolean** *DUMMY4*;

**procedure** *MEP* ($a$, $b$, $c$, $x1r$, $x1i$, $x2r$, $x2i$, *B*, *EXIT*);
{Itt következik az (5.14) eljárásdeklaráció folytatása, melynek ismételt felírásától most eltekintünk.} **end** *MEP*;
$x:=xE:=p:=pE:=0.55$; $i:=1$; (5.16)
*ISM* : *MEP* ($p$, *sqrt* $(1+p)$, $-1.1046$, $x$, *DUMMY1*,
*DUMMY2*, *DUMMY3*, *DEFEKT*);

$pE:=p$; $xE:=x$; $i:=i+1$; **go to** *ISM*;
*OUT*: {$i$, $x$, $p$ kinyomtatása}; {Megállás};
*DEFEKT* : {Megállás} **end**

Az (5.16) program szerkezete eléggé magától értetődő, és így azzal nem kell részletesen foglalkoznunk. Annyit talán érdemes megjegyeznünk, hogy az adott esetben célszerűnek mutatkozott a **for** ciklus mellőzése, és a ciklusnak feltételes utasítás segítségével való programozása. Az eljárásutasítások aktivizálása az előzők ismeretében érthető: a két eljárásutasítás közül az első az $x$, a második a $p$ változóba helyezteti a másodfokú egyenlet megoldását, és ez éppen megfelel a leírt iterációs algoritmus követelményeinek. Az aktuális paraméterek, amelyekkel a formális $a$, $b$ és $c$ para-

* Ez csak illusztratív feladat; a fenti egyenletrendszert ui. a leírt iteráció segítségével egy gyakorlott kalkulátor asztali számológépen egy óra alatt megoldja.

métereket helyettesítjük, aritmetikai kifejezések, és e paraméterek érték szerinti behívása most valóban gépidő-takarékosságot eredményez.

Külön kell szólnunk a *DUMMY*1, ..., *DUMMY*4 változókról, melyekre látszólag semmi szüksége nincs a programnak, mégis jelen vannak. Ennek a programnak valóban nincs szüksége ezekre a változókra, azonban a *MEP* eljárás használhatósága érdekében mégis definiálni kell őket. Kész eljárásoknak valamely konkrét programba való beépítése közben gyakran előfordul, hogy az adott esetben nem használjuk fel mindazt a kimenő információt, amit a beépítendő eljárás nyújt. Az aktuális és formális paraméterek számának azonban ekkor is meg kell egyeznie, ellenkező esetben a fordítóprogram le sem fordítja a programot. Ilyenkor a programozó kénytelen néhány olyan azonosítót definiálni, amelyet azután a program nem fog érdemben használni. Ilyen „helykitöltő" változók a *DUMMY*1 stb., amelyek egyetlen szerepe, hogy összeférhetővé tesznek az adott programot az (5.14) eljárásdeklarációval. Ugyancsak helykitöltő most a *DEFEKT* címke, amelyre soha sem kerül rá a vezérlés, mert az iteráció során nem kerülhet sor értelmetlen egyenlet megoldására.

Ez a példa arra figyelmeztet, hogy a programozónak mindig pontosan ismernie kell a programba beépítendő „idegen" eljárásnak, ha nem is a működését, de legalább formális azonosítóinak szerepét, és az eljárás alkalmazására vonatkozó kikötéseket. Ezért rendkívül fontos, hogy az irodalmilag publikált eljárások mindig tartalmazzanak egy olyan kommentárt, amely felvilágosítja a fentiekről a programozókat.

Következő példaként vizsgáljuk az alábbi feladatot: adott egy $A$ mátrix, melynek 7 sora és 10 oszlopa, valamint egy $B$ mátrix, melynek 10 sora és 7 oszlopa van, egy hételemű $v$ vektor és egy $7 \times 7$ típusú $C$ mátrix. El kell végeznünk a következő mátrix-műveleteket:

1. Összeszorzandó az $A$ és $B$ mátrix;

2. ezután megoldandó az $ABx = v$ lineáris algebrai egyenletrendszer;

3. a kapott $x$ vektor összeszorzandó a $C$ mátrixszal.

A feladat céljaira először két eljárásdeklarációt írunk fel, egyet mátrixok szorzatának meghatározására, és egyet lineáris algebrai egyenletrendszerek megoldására. Ezeket általános formában alakitjuk ki, hogy más program is felhasználhassa őket.

A mátrixok szorzatát előállító eljárásdeklaráció a következő lehet:

```
procedure MPRODUCT(A, B, C, n, m, r);
comment A MPRODUCT eljárás az n × m típusú A mátrix
szorzatát képezi az m × r típusú B mátrixszal, és az eredményt elhelyezi az n × r típusú C mátrix elemeiben;
value n, m, r; integer n, m, r; array A, B, C;
begin integer i, j, k; real S;
   for i:= 1 step 1 until n do                                (5.17)
   for j:= 1 step 1 until r do
      begin S:=0; for k:=1 step 1 until m do
         S:= A[i, k] × B[k, j] + S;
      C[i, j]:= S
      end j és i; end MPRODUCT;
```

Az eljárásnak hat formális paramétere van: a három mátrixnév és a három rendszám. A rendszámokat érték szerint behivottaknak választjuk, mert ezzel gépidő-takarékosságot érünk el. Itt rögtön megjegyezzük, hogy az ALGOL nyelv tömbök érték szerinti behivását is megengedi. Ebben az esetben — a skaláris változók érték szerinti behivásának analógiájára — az eljárás aktivizálását értékadó utasitások előzik meg, amelyek az aktuális paramétertömb elemeit áthelyezik a formális paramétertömb elemeibe. (A fordítóprogamnak ilyenkor egy annyiszorosan egymásba skatulyázott ciklust kell generálnia, amilyen mértékűek a formális, ill. aktuális paramétertömbök, mivel a tömbelemek átvitele nyilván ciklusban történik.) Ha egy eljárásnak valamely tömbparamétere érték szerint kerül behivásra, akkor azonosítóját szintén fel kell venni a **value**-listára, a többi érték szerint behivott paraméter azonosítójával együtt. A **value**-listán nincsenek megkülönböztetve a tömbazonosítók a skaláris változók azonosítóitól.

Eljárások valós típusú tömbparamétereit a specifikációs részben egy **array** kezdetű listán [lásd (5.17)-et] kell felsorolni. Az **array**-lista formailag is különbözik a tömbdeklarációktól (IV. fejezet 3. p.), mivel a listán csak azonosítók szerepelnek, korlátpár nélkül. A tömbök méretszáma megállapítható az eljárástörzsben az illető tömb elemeire hivatkozó utasításokból: például (5.17)-ben az eljárástörzsből látszik, hogy itt *A*-nak, *B*-nek és *C*-nek kétméretű tömbnek kell lennie. Az ALGOL szintaktikus szabályai úgy rendelkeznek, hogy az aktuális paramétertömböknek mindig ugyanolyan méretszámúnak kell lenniük, mint a formális paramétertömböknek, mely utóbbiak méretszáma az eljárástörzsből mindig megállapítható. Az indexek értékhatáraira nézve pedig mindig az aktuális paramétertömb értékhatárai az irányadók. Ezért nincs értelme értékhatárokat megjelölni az eljárás specifikációs részében. A specifikációs résznek meg kell jelölnie a formális paramétertömb típusát: a tömbdeklarációkhoz hasonlóan, a logikai típusú tömbök egy külön **Boolean array**, az egész típusú tömbök pedig egy **integer array** kezdetű listán sorolandók fel. Akárcsak a tömbdeklarációk esetében, a specifikációs részben is megengedett, de nem kötelező az **array** alapjel elé a **real** kiírása, amennyiben valós típusú tömbök specifikálásáról van szó.

Annak, hogy az (5.17) eljárásban a formális paramétertömböket név és nem érték szerint hívtuk be, az az oka, hogy érték szerinti behívás esetén a célprogramnak külön memóriateret kell fenntartania az érték szerint behívott tömbparaméterek részére, holott azok egyszer már úgyis el vannak helyezve. Tömbök érték szerinti behívása nélkül sem veszítünk az ALGOL hatékonyságából, és ezért sok gépi reprezentáns ezt a lehetőséget eleve kizárja.

Eredetileg kitűzött feladatunkra visszatérve, szükségünk van még egy eljárásdeklarációra, amely lineáris algebrai egyenletrendszereket old meg. Ennek az eljárásdeklarációnak a céljára felhasználhatjuk a már többször vizsgált (3.78) algoritmust (lásd IV. fejezet 8. p.). Az algoritmust eljárásdeklarációvá kiegészítve, a következőt kapjuk:

```
procedure LEMP (a, n); vaule n; integer n; array a;
begin integer i, j, k; real T, m;
for i:=1 step 1 until n do begin                    (5.18)
    m:=1/a[i, i];
    for k:=i+1 step 1 until n+1 do a[i, k]:=m×a[i, k];
    for j:=1 step 1 until n do if j≠i then begin T:=a[j, i];
        for k:=i+1 step 1 until n+1 do a[j, k]:=a[j, k] -
        T×a[i, k]
    end if és j;
end i end LEMP;
```

Most már rendelkezésünkre állnak a feladat megoldásához szükséges eljárások. Lehetne még egy eljárást felírni mátrixnak vektorral való szorzására, de ez mellőzhető, mert az *MPRODUCT* eljárás megfelel erre a célra is, ha $r=1$ értékkel aktivizálják. Most tehát egy vezérprogramot kell még szerkesztenünk, amely megfelelően aktivizálja ezeket az eljárásokat. Ez a vezérprogram — a szokott jelölések felhasználásával — a következő lehet:

```
begin array A[1:7, 1:10], B[1:10, 1:7], v[1:7];
   C[1:7, 1:7], AA[1:7, 1:8], z, x[1:7, 1:1]; integer i, j;
   procedure MPRODUCT (A, B, C, n, m, r);
{az (5.17) eljárásdeklaráció folytatása}; end MPRODUCT;
   procedure LEMP (a, n);
      {az (5.18) eljárásdeklaráció folytatása}; end LEMP;
{A és B beolvasása};
MPRODUCT (A, B, C, 7, 10, 7);
{v vektor beolvasása};                              (15.19)
for i:=1 step 1 until 7 do begin
   for j:=1 step 1 until 7 do AA[i, j]:=C[i, j];
   AA[i, 8]:=v[i]; end i;
LEMP (AA, 7);
   for i:=1 step 1 until 7 do x[i, 1]:=AA[i, 8];
   MPRODUCT (C, x, z, 7, 7, 1);
{eredmények felhasználása vagy kinyomtatása}; {Megállás}
end
```

Az (5.19) programmal kapcsolatban ismét van néhány érdekes megjegyeznivaló. Az első, ami szembetűnő, hogy a programban deklarált egyes tömbök azonosítója megegyezik olyan tömbök azonosítójával, amelyek formális paraméterei olyan eljárásoknak, amelyek ugyanebben a programban vannak deklarálva. Ilyenek az *A*, *B* és *C* tömbök azonosítói. Ez tökéletesen összefér az ALGOL 60 szintaktikus szabályaival, és mivel az újradeklarálás fogalma már ismert számunkra (lásd IV. fejezet 2. p.), ezt nem is érezzük meglepőnek. Itt is az a helyzet, hogy az eljáráson belül a formális paraméter azonosítója bizonyos értelemben lokális jelentésű, és a fordítóprogram automatikusan gondoskodik a vezérprogram és az eljárások azonosítóinak megkülönböztetéséről.* Azonban az azonosítók ilyenfajta összekeverése általában rontja a programok áttekinthetőségét, és ha a formális szabályokkal nem is, a programozás „illemszabályaival" ellenkezik.**

Az, hogy bizonyos betűket (*i*, *j*) mind a vezérprogram, mind annak deklarált eljárásai felhasználnak lokális azonosítóként, sokkal magától értetődőbb, és nem ellenkezik semmilyen írott vagy íratlan szabállyal, sőt elkerülése nem is lenne célszerű, mert nagyobb programok esetén jókora szerkesztői munkát róna a programozóra.

Az (5.19) program esetében ismét azt tapasztaljuk, hogy a felhasznált eljárások „összeillesztése" érdekében az eljárásutasítások közé adminisztratív programrészeket kell iktatnunk, amelyek előkészítik a bemenő adatokat a soron következő eljárás részére. Programunkban többek között össze kellett raknunk a *C* mátrixot és a *v* vektort egyetlen $7 \times 8$-as típusú mátrixszá, mert a lineáris egyenletrendszermegoldó program *(LEMP)* ilyen formában követeli a be-

* Az eljárásdeklarációk által felhasznált azonosítók osztályozására az 5. p. végén külön kitérünk.

** Ilyen „illemszabályok" a közönséges matematikai képletírásban is vannak, pl. az $f(x)$ függvény primitív függvényét nem illik (bár szabad) ilyen alakban felírni:

$$F(x) = \int_0^x f(x)\,dx.$$

menő információt. A gyakorlati programozás során a programozó általában szívesebben vállalja ilyen adminisztratív programok beiktatását, minthogy belenyúljon egy kész és kipróbált eljárásba, és az adminisztráció elkerülése érdekében átdolgozza. Ez utóbbi esetben ugyanis az eljárást újra ki kell próbálnia, és meg kell győződnie arról, hogy az átdolgozás közben nem követett-e el valamilyen hibát. A fentiek természetesen arra az esetre vonatkoznak, amikor egy „kész" (pl. irodalmilag publikált) eljárást veszünk át valahonnan. Ha az eljárásdeklarációt a programmal egyidőben, egyazon személy dolgozza ki, akkor írható oly módon, hogy a lehető legjobban megfeleljen az őt aktivizáló programnak. Azonban „közérdekű" eljárás esetén még ilyen esetben is megfontolandó, hogy ez nem rontja-e az általános felhasználhatóságot.

Az eddigi példákban szereplő eljárásdeklarációk [(5.2), (5.14), (5.17), (5.18)] speciálisak voltak abból a szempontból, hogy az eljárástörzsben mindössze kétféle azonosító szerepelt: formális paraméter vagy az eljárástörzsre nézve lokális azonosító. Ez nem törvényszerű; egy ALGOL-eljárás törzse hivatkozhat minden olyan azonosítóra, amely abban a blokkban, amelyben az eljárás deklarálva van, érvényes deklarációval rendelkezik, így akár azokra az azonosítókra, amelyek ugyanabban a blokkban vannak deklarálva, mint maga az eljárás, akár azokra, amelyek egy magasabb rendű blokkban vannak deklarálva. Hangsúlyozzuk, hogy ez mindenfajta azonosítóra érvényes, így többek között más eljárások azonosítójára* is: egy eljárás behívhat egy olyan másik eljárást, amely akár ugyanabban, akár magasabb rendű blokkban van deklarálva. Az is megengedett, hogy az eljárástörzs olyan azonosítókra hivatkozzék, amelyek deklarációja sorrendben később következik; a szintaktikus szabályok megengednék például, hogy az (5.19) programban az *MPRODUCT* eljárás behívja a *LEMP* eljárást annak ellenére, hogy ez utóbbinak a deklarációja sorrendben *MPRODUCT* deklarációja után következik. Hasonló érvényes azokra a változókra is, amelyek deklará-

* Az 5. p.-ban látni fogjuk, hogy az eljárás saját azonosítójára is.

ciója esetleg az őrájuk hivatkozó eljárás deklarációja után következik.

Közérdekű, irodalmilag publikálandó eljárások esetén azonban felettébb célszerűtlen olyan azonosítókra hivatkozni, amelyek globálisak az eljárás törzsében, mert ez bizonyos megkötést jelent az eljárást aktivizáló vezérprogram azonosítórendszerére nézve. Ha mégis ezt teszszük, pontos és részletes kommentárt kell az eljáráshoz fűznünk, amely tüzetesen leírja, hogy milyen globális azonosítói vannak az eljárásdeklarációnak, hogyan kell őket deklarálni stb. Különösen vonatkozik ez az eljárás által behívott más eljárások azonosítójára.

Más a helyzet „házi használatra" készült eljárások esetén. Ezekben sok esetben nagyon is hasznos globális azonosítók szerepeltetése, mert általa gépidő- és memóriatakarékosság érhető el. Tegyük fel például, hogy az (5.2) eljárásdeklarációt egy olyan program aktivizálja, ismételten a program különböző helyein, amely az átalakítandó komplex szám abszolút értékét minden esetben ugyanazon *alfa*, fázisszögét ugyanazon *beta* változóban képezi, és az eredményt is mindig ugyanazon, mondjuk *u* és *v* változóknak kell átadni. Ebben az esetben az *ÁTALAKÍTÁS* eljárást definiálhatjuk formális paraméter nélküli eljárásként is, és eljárástörzsének valamennyi azonosítója globális lesz az eljárástörzsre nézve (ugyanakkor lokális az eljárást aktivizáló blokkra nézve). Egy ilyen program sémája a következő:

```
begin real u, v, alfa, beta;                                    (5.20)
   procedure ÁTALAKÍTÁS; begin u:=alfa × cos (beta);
   v:=alfa × sin (beta); end ÁTALAKÍTÁS;
{utasítások, amelyek értéket adnak alfának és betának};
ÁTALAKÍTÁS;
{utasítások, amelyek felhasználják u-t és v-t, és újabb értéket adnak alfának és betának};
ÁTALAKÍTÁS;
{utasítások, amelyek felhasználják u-t és v-t} end
```

Az (5.20) konstrukció előnye, hogy megtakarítjuk általa az aktuális és formális paraméterek cseréjét, aminek következtében mind térben, mind időben rövidebb programhoz jutunk, mintha az (5.2) alakban megadott eljárásdeklarációt használnánk. Ebben az esetben az eljárás és az őt aktivizáló program kapcsolatát nem formális (aktuális) paraméterek segítségével biztosítjuk, hanem az eljárástörzs globális változói segítségével. Ez azonban maga után vonja azt, hogy nem tudjuk a program különböző helyein más és más aktuális változókkal aktivizálni az eljárást, tehát ha a programozás során ennek szükségessége is felmerül, akkor vissza kell térni az (5.2) alakra, esetleg érték szerint behívott bemenő paraméterekkel. Éppen ennél az oknál fogva irodalmilag publikálandó eljárásokban globális azonosítók alkalmazása nem ajánlható: nem tudjuk az eljárást különböző helyeken más és más nevű aktuális azonosítókkal aktivizálni, hanem kötve vagyunk az eljárástörzsben szereplő globális azonosító nevéhez.

Következő példánkkal* azt illusztráljuk, hogy a név szerinti paraméterbehívás néha egészen ügyes programozási fogásokat tesz lehetővé. E fogások alapgondolata az, hogy formálisan változónak specifikált (**real**, **integer** vagy **Boolean** típusú) formális paraméter tetszőleges aritmetikai (logikai) aktuális paraméterrel kicserélhető az eljárás aktivizálásakor. Ez lehetőséget ad első ránézésre látszólag értelmetlen eljárásdeklarációk felírására, amelyek akkor válnak értelmessé, ha melléjük helyezzük az őket aktivizáló eljárásutasítást is.

Tekintsük a következő eljárásdeklarációt:

```
procedure SPRODUCT(a, b, n, i, S); value n;
   integer n, i; real a, b, S;
   begin S:=0; for i:=1 step 1 until n do
   S:=S+a×b end SPRODUCT;
```
(5.21)

Az (5.21) eljárásdeklaráció azonosítója a *Scalar PRODUCT* (skalárszorzat) rövidítése, azonban első pillanatban

* A példát az ALGOL-jelentésből vettük át.

nem látszik, hogyan lehet ennek az eljárásnak a segítségével két vektor skaláris szorzatát előállítani. Tegyük fel, hogy c két vektor $U$ és $V$, rendszámuk (elemeik száma) 10. Ekkor (5.21) a következő eljárásutasítással aktivizálható (feltesszük, hogy $k$ az aktivizáló program **integer** típusú, $X$ pedig **real** típusú deklarált azonosítója):

$$SPRODUCT(U[k], V[k], 10, k, X); \qquad (5.22)$$

Ha végiggondoljuk, hogy az (5.8) kapcsán leírt behelyettesítési séma alkalmazásakor mit hajt végre a gép az (5.21) eljárásdeklaráció eljárástörzsén, amikor az (5.22) eljárásutasítást fordítja, akkor látjuk, hogy pontosan kialakul az a program, amely az $U$ és $V$ vektorok skalárszorzatát átadja az $X$ változónak [vö. (3.48)-cal a III. fejezet 7. p.-ban!]. Ez a fogás minden szempontból tökéletesen korrekt, $U[k]$ és $V[k]$, mint indexes változók, aritmetikai kifejezések, tehát megadhatók aktuális paraméterként az (5.21) eljáráshoz, mivel annak megfelelő formális paraméterei **real** típusúnak vannak specifikálva. Azért tűnik mégis szokatlannak, mert az eljárás aktivizálásakor az aktuális paraméterkifejezéseknek nincs aktuális értékük, hanem csak az eljárástörzs végrehajtása közben kapnak majd értéket, amikor már $k$-nak meghatározott értéke lesz. Éppen ez oknál fogva az egész program csak abban az esetben értelmes, ha — mint jelen esetben — $a$ és $b$ név szerint behívott paraméterek; ha érték szerint behívottak lennének, akkor az egész program teljesen értelmetlen és szemantikusan hibás volna. Ez tehát példa arra, hogy egy program teljesen értelmetlenné válhat, ha egy paraméterét név szerint behívottról érték szerint behívottra (vagy fordítva) változtatjuk.

Az (5.21) eljárásdeklaráció nemcsak vektorok skaláris szorzatának előállítására alkalmas. Megfelelő aktivizálás esetén bármiféle szorzatösszeget kiszámíttathatunk vele. Aktivizáljuk pl. (5.21)-et a következő szumma kiszámítására:

$$Q = \sum_{k=1}^{N} \sin(kx)e^{-kx} \quad (x \text{ rögzített}). \qquad (5.23)$$

Ekkor a megfelelő eljárásutasítás a következő lehet:

$$SPRODUCT(sin\,(k \times x), exp\,(-k \times x), N, k, Q); \quad (5.24)$$

Az aktuális paramétereknek az eljárástörzsbe való behelyettesítése után éppen az (5.23) szumma kiszámításának programja adódik.

Ha a (3.48) programot a lehető legkézenfekvőbb módon egészítenénk ki eljárásdeklarációvá — hasonlóan a (3.78) programhoz [lásd az (5.18) eljárásdeklarációt] —, akkor a kapott program mindezen fogások véghezvitelére teljesen alkalmatlan lenne. Ez arra int, hogy mielőtt egy eljárást végleges formában megírnánk, alaposan meg kell gondolnunk, milyen célra kívánjuk azt használni, és milyen jellegű aktivizálásokra számíthatunk.

Eljárásdeklarációkkal és eljárásutasításokkal kapcsolatos programozási fogások illusztrálására bőven lehetne további példákat felhozni, mert az ALGOL nyelv tág lehetőségeket biztosít ilyenekre. Némi programozási gyakorlat birtokában a programozó magától is rájön hasonlókra, ezért a példák sorát nem folytatjuk, hanem áttérünk néhány további szintaktikus és szemantikai szabály tárgyalására.

Az eddigiekben példákat láttunk skaláris változó (**real**, **integer** és **Boolean**), tömb (**array**) és címke (**label**) jellegű formális paraméterekre. A zárójelben álló szóalapjel az illető formális paramétertípus specifikációs listájának jellemző alapjele. A fentieken kívül lehet egy eljárás paramétere kapcsoló (a specifikációs listát bevezető alapjel **a switch**), eljárás (**procedure**), függvényeljárás (**real procedure**, **integer procedure**, **Boolean procedure**) vagy idézet (**string**). Eljárás jellegű paraméterek alkalmazására a következő pontban látunk majd példákat, az idézetek szerepéről a 6. p.-ban lesz szó. Kapcsoló jellegű paraméterek alkalmazására e lehetőség kisebb jelentősége miatt — nem adunk példát.

Az alábbiakban egy fontos szemantikai szabályt ismertetünk, amely az eljárások törzsében szereplő azonosítók hatásköréről rendelkezik. E szabály szerint egy eljárás aktivizálásakor az aktivizált eljárás törzse oly módon helye-

zendõ be a behívó programba, hogy köré egy fiktív blokk képzelendő, amelynek az érték szerint behívott formális paraméterek lokális azonosítói, és amely magában foglalja az érték szerint hívott paraméterekre vonatkozó előzetes értékadásokat. A **value**-listán szereplő formális paraméterek, definíciószerűen, globálisak az eljárástörzsre nézve, de lokálisak a fentiek szerint definiált fiktív blokkban. Ezt a fiktív blokkot — ugyancsak definíciószerűen — még abban az esetben is jelenlevőnek kell tekinteni, ha az eljárásnak nincsenek érték szerint hívott formális paraméterei. Ugyancsak független e fiktiv blokk jelenléte attól is, hogy maga az eljárástörzs formálisan blokk-e vagy sem.

A fenti szabály illusztrálására térjünk vissza az (5.15) transzcendens egyenletrendszer megoldásának feladatához. [A megoldóprogramot lásd (5.16), a felhasznált eljárást (5.24) alatt.] Az elmondott szemantikai szabályok értelmében az *ISM* címkével ellátott eljárásutasítás hatásában a következővel egyenértékű:

```
ISM : begin real a, b, c;                                        (5.25)
a:=p;  b:=sqrt (1+p);  c:=-1.046;
begin real M, P;  if a=0 ∧ b=0 then go to DEFEKT;
   {(5.14) eljárástörzsének utasitásai};  end MEP
end fiktiv blokk;  {folytatás}
```

Mint látható, az eljárás aktivizálásakor az *ISM* címkéjű eljárásutasitás helyére egy kétszeresen egymásba skatulyázott blokkrendszer lép. A külsõ blokk lokális azonositóként deklarálja az érték szerint hívott három formális paramétert, *a*-t, *b*-t és *c*-t. Ezután következnek az ily módon deklarált paraméterváltozókra vonatkozó értékadások, majd maga az eljárástörzs, amely példánkban formálisan is blokk. Ha az eljárástörzs maga nem blokk, akkor a fenti szabály értelmében egy egyszeres blokk kerül a behivó programba. Ha például az (5.2) eljárásdeklarációt érték szerint aktivizáljuk, a behivó programba a következő utasitáscsoport kerül be:

```
begin real ro, fi;                                                    (5.26)
   ro := alfa;  fi := beta;
   begin  u := ro × cos(fi);  v := ro × sin(fi);  end  eljárás-
   törzs
end fiktív blokk;  {folytatás}
```

Az (5.25) és (5.26) példák alapján nyilvánvaló, hogy az idézett szabály — kissé bonyolult megfogalmazásban — azt fejezi ki, hogy az érték szerint hívott formális paraméterek lényegében lokálisak az eljárásra nézve, és azokra „kívülről" nem lehet hivatkozni. (Benne van továbbá az a szabály is, hogy az említett értékadások az eljárástörzs végrehajtása előtt kerülnek sorra — amint erről a 2. p.-ban is volt már szó.)

Hátra van még annak a kitételnek a magyarázata, amely szerint az eljárástörzset körülölelő fiktív blokk akkor is jelen van, ha érték szerint hívott formális paraméterek egyáltalában nincsenek. Ez a kitétel az eljárástörzs címkéinek lokalitását védi, és lényegében azt jelenti, hogy sem az eljárástörzs kezdetére, sem annak belsejébe kívülről **go to** utasítással nem szabad ráugratni, még akkor sem, ha az eljárásnak nincsenek érték szerint hívott paraméterei. Mindez következménye a címkék hatásköréről a IV. fejezet 5. p.-jában mondottaknak. Ugyanezek a szabályok megengedik viszont, hogy egy vezérlésátadó utasítás kivezessen az eljárástörzsből, mert ilyenkor a vezérlés de facto egy magasabb rendű blokkba kerül.

A fentiek talán megvilágítják, miért kellett a IV. fejezetben olyan részletesen foglalkozunk az egymásba skatulyázott blokkokkal kapcsolatban felmerülő szemantikai kérdésekkel. Ugyanis minden olyan esetben, amikor egy blokk törzsében egy eljárás aktivizálódik, a program végrehajtása közben egymásba skatulyázott blokkok jönnek létre, és ezek szemantikája csakis a IV. fejezetben mondottak ismeretében válik érthetővé. (Például, ha az eljárástörzs valamelyik lokális azonosítója megegyezik az eljárást aktivizáló blokk egy azonosítójával, akkor létrejön a változó újradeklarálása, és érvénybe lépnek az újradeklarálással kapcsolatos szemantikai szabályok stb.)

Eddigi valamennyi példánkban azt láttuk, hogy az eljárások behívásában az aktuális és formális paraméterek típusa megegyezett. Bizonyos esetekben nem okozna zavart, ha meg lenne engedve a formális paraméter specifikált típusától különböző típusú aktuális paraméter alkalmazása. Például általában szintaktikusan és szemantikusan korrekt ALGOL-programra vezet, ha valósnak specifikált formális paraméterre egész típusú aktuális paraméterkifejezést adunk meg. Azonban e kérdés szemantikája még a módosított jelentésben sincs véglegesen tisztázva, csak egy utalás van arról, hogy a deklaráció és specifikáció összeférhetőségének kérdését a későbbiekben fogják tisztázni.

Eljárások aktivizálásával kapcsolatban felvetődik még egy szemantikai probléma. Előfordulhat, hogy egy név szerint behívott formális paraméter azonosítója megegyezik az eljárástörzs valamelyik lokális azonosítójával. Ilyenkor az eljárástörzs „kitöltése" után olyan program jön létre, amelyben ugyanazt az azonosítót két különböző célra használják, ami nyilvánvaló szemantikai kétértelműségekre vezethetne. Az ALGOL-jelentés úgy rendelkezik erről az esetről, hogy ilyenkor a fordítóprogram köteles ezt az eljárástörzs azonosítórendszerének szisztematikus megváltoztatása útján kiküszöbölni. Ez nagyon hasonló a címkék hatáskörével kapcsolatban tárgyalt egyik szabályhoz [lásd a (4.30) kapcsán elmondottakat].

A tapasztalat azt mutatja, hogy a gyakorlatlan programozók kissé tanácstalanok, amikor egy eljárásdeklarációt kell megírniuk, mert nem tudják, hogy a rendelkezésre álló sok lehetőség közül melyiket válasszák, hogyan szervezzék az eljárást, milyenek legyenek a paraméterek stb. Az előzőkben nagyrészt elmondottuk a szempontokat, de talán nem árt, ha még egyszer összefoglaljuk azokat:

1. Az eljárás formális paraméterei az eljárás és az őt behívó program kapcsolatának fenntartására szolgálnak. Tehát formális paraméternek azokat az azonosítókat választjuk, amelyekkel vagy kiinduló információt adunk, vagy kimenő információt közlünk. (Természetesen az információközlés nem mindig értékadás: pl. kimenő információ

az is, ha az eljárás a „vészkijáratként" fenntartott címke-paraméterre adja a vezérlést.)

2. Az előző pont alapján világos, hogy azokat az azonosítókat, amelyeknek nincs kapcsolatuk a Behívó programmal, az eljárástörzsben lokálisan deklaráljuk.

3. Érték szerint hívjuk azokat a paramétereket,

*a)* amelyekről feltehető, hogy az eljárás aktivizálása során aktuális paraméterként kifejezésparaméterek fognak megjelenni, továbbá

*b)* ezeknek a kifejezéseknek meghatározott értékük lesz az eljárás behívásának pillanatában, és

*c)* ez az érték az eljárástörzs végrehajtása közben nem változik meg.

Magától értetődik, hogy érték szerint hívni csak olyan paramétert lehet, melyre van értelme értékéről beszélni.

4. Név szerint hívjuk azokat a paramétereket, amelyeknek nincs értékük definiálva, vagy azokat, amelyekre a harmadik pont valamelyik feltétele — különös tekintettel a *b)* és *c)* feltételekre — nem teljesül.

5. Alkalmazhatunk eljárások törzsében globális azonosítókat is, melyek szintén alkalmasak arra, hogy általuk kapcsolatot létesítsünk az eljárástörzs és a behívó program között. Ezt a módszert azonban irodalmilag publikálandó eljárások esetében ajánlatos kerülni.

6. A formális paraméterek közül legalábbis az érték szerint behívottak specifikálásáról nem szabad megfeledkeznünk. Ez helyettesíti a deklarációt; a formális paraméterek ui. nem esnek deklarációs kötelezettség alá. Ha tehát egy teljes ALGOL-programban deklarálatlan azonosító fordul elő, akkor ez

*a)* vagy standard eljárás azonosítója,*

*b)* vagy címke,

*c)* vagy név szerint behívott formális paraméter.

Minden egyéb azonosítót deklarálni kell.

* Azért mondtunk standard eljárást, és nem standard függvényt, mert a gépi reprezentánsok a II. fejezet 5. p.-ban bevezetett standard függvényeken kívül általában egy sereg további standard eljárást is definiálnak, melyek szintén behivhatók külön eljárásdeklaráció nélkül.

## 4. Függvényeljárások

A II. fejezet 5. p.-ból ismerjük a standard függvények fogalmát. Ezek a matematikai analízis, valamint az elemi számelmélet leggyakrabban előforduló függvénykapcsolatai, melyek az ALGOL hivatkozási nyelv feltételezése szerint minden elektronikus számítógép memóriájában — előre kidolgozott program formájában — tárolva vannak, és szükség esetén bármikor aktivizálhatók, tehát minden ALGOL fordítóprogram elő tudja hívni őket, ha standard azonosítójukkal találkozik a lefordítandó programban. Ez úgy értendő, hogy a fordítóprogram olyan utasításokat helyez el a célprogramban, amelyekre a gép memóriájának abból a részéből, amelyben a standard függvények előállítási programjai tárolnak, a megnevezett program átkerül azokba a memóriarekeszekbe, ahol a célprogram tárol, és (amennyiben szükséges) végrehajtódnak rajta azok az átalakítások, amelyek működtetéséhez szükségesek.

A standard függvények köre azonban igen szűk, és bár egyes gépi reprezentánsok ezt jelentősen kibővítik az ALGOL-ban definiált függvényekhez képest, a gyakorlati programozásban mégis gyakran felmerül a szükségessége olyan speciális függvények használatának, amelyek nem szerepelnek a standard függvények listáján, mely a memória véges kapacitása miatt nyilván nem bővíthető a végtelenségig. Ezért nagyon kívánatos, hogy ALGOL-program segítségével definiálhatók legyenek nem standard függvények. Erre lehetőséget nyújtanak természetesen az ALGOL eddig megismert részei is, ugyanis az említett kívánalom kielégíthető egy eljárásdeklarációval, amelynek bemenő paraméterei a függvény argumentumai, kimenő paramétere pedig a függvényérték. Az egyetlen kellemetlenség, hogy egy ilyen eljárás aktivizálásához külön eljárásutasítást kell beiktatni, és az eljárás aktivizálása nem építhető bele pl. egy aritmetikai kifejezés kiszámításába, amint az a standard függvények esetében történik. A matematikai képletíráshoz való közelítés szempontjából pedig kívánatos, hogy a speciális függvények ALGOL-programjai, ha már egyszer definiálva vannak, ugyanúgy legyenek aktivizálhatók, mint

a standard függvények. Tehát például ahogyan felírhatunk egy aritmetikai kifejezést

$$x+(1-sin\,(x))\times exp\,(x)$$

alakban, ugyanúgy írhassunk például

$$x+(1-J0(x))\times K2(x)$$

alakot is, ha már előzetesen definiáltuk a $J0$ és $K2$ Bessel-függvényeket előállító ALGOL-programokat.

A fenti követelmények kielégítésére vezették be az ALGOL-ban az ún. *függvényeljárások* fogalmát. A függvényeljárások igen hasonlók a már megismert közönséges eljárásokhoz, csupán a közönséges eljárásoktól eltérően, a függvényeljárás azonosítójához minden egyes aktivizálás hozzárendel egy értéket, amelyet *függvényértéknek* nevezünk. A függvényeljárások deklarációjának legelső alapjele mindig egy típusdeklaráló alapjel (**real, integer** vagy **Boolean**), amely meghatározza az eljárás aktivizálásakor keletkező érték típusát. Ahhoz, hogy az eljárás által definiált függvényérték meg legyen határozva, kell az eljárásdeklarációban legalább egy olyan értékadó utasítás, amelynek bal oldalán az eljárás saját azonosítója áll, és amely az eljárás aktivizálása során végre is hajtódik. Az az érték lesz a függvényérték, amelyet a legutoljára végrehajtott ilyen értékadó utasítás ad az eljárás azonosítójának. Ha egy eljárásdeklaráció eleget tesz a fenti feltételeknek, akkor az eljárás aktivizálása teljesen a standard függvények analógiájára történhet, tehát az eljárás azonosítója után kerek zárójelben — egymástól vesszővel elválasztva — felsoroljuk az aktuális argumentumokat, és ezt az egész konstrukciót használhatjuk aritmetikai (logikai) kifejezések alkatrészeként, amint azt a bevezetőben mutatott példán láttuk.

Például a sinus hiperbolicus függvényt a következő eljárásdeklarációval deklarálhatjuk:

```
real procedure sh (x); value x; real x;
    sh:=(exp(x)-exp(-x))/2;                    (5.27)
```

Látjuk, hogy ez formailag csupán annyiban tér el az előzőkből már ismert eljárásdeklarációktól, hogy első alapjele

a **real**, amely meghatározza a függvényérték típusát, és az egyetlen értékadó utasításból álló eljárástörzs bal oldalán *sh* azonosítója áll, mégpedig az aktuális argumentumok nélkül. Jól jegyezzük meg: amikor az eljárás aktuális értékét meghatározó értékadó utasításokat írjuk, a megfelelő *értékadó utasítás bal oldalán az eljárás azonosítója aktuális argumentumok nélkül áll.*

Az előző pontokban az eljárásdeklarációkkal kapcsolatban ismertetett minden szabály (formális-aktuális paramétercsere szabályai, név és érték szerinti behívás, specifikálási szabályok stb.) érvényes a függvényeljárásokra is. A paraméterek típusának megválasztásában azok a szempontok az irányadók, amelyeket az előző fejezet végén felsoroltunk. Az a tény, hogy (5.27) eljárástörzse egyetlen utasításból áll, a példa egyszerűségének következménye, semmiképpen sem általános szabály. A paraméterek száma — akárcsak egyéb eljárások esetén — itt is tetszőleges, tehát mód van többváltozós függvények definiálására is.

Ha a programban valahol az (5.27) eljárásdeklarációval definiáltuk a *sh*(*x*) függvényt, ezután már használhatjuk az *sh* azonosítót aritmetikai kifejezésekben. Például a következő aritmetikai kifejezést:

$$\sin x + e^x \operatorname{sh}^2 x + \cos(2kx)$$

felírhatjuk az *sh* azonosító közvetlen felhasználásával:

$$sin(x) + exp(x) \times sh(x) \uparrow 2 + cos(2 \times k \times x)$$

Itt megjegyezzük, hogy a standard függvényekkel megegyezően egy függvényeljárás azonosítója az aktuális argumentumok listájával együtt — ezt az együttest az ALGOL-ban *függvénykifejezésnek* nevezzük — korrekt aritmetikai kifejezés, és mint ilyen, megadható aktuális paraméterként olyan helyeken, ahol az ALGOL egyéb szabályai kifejezésparamétereket megengednek.

További példaként függvényeljárás deklarálására és aktivizálására készítsünk ALGOL-programot egyszeres szummák kiszámítására. Legyen az eljárás azonosítója *SUM*, és képezzük ki az eljárást függvényeljárásként. Az eljárás deklarációja a következő lehet:

```
real procedure SUM (i, n, y); value n;
integer i, n; real y;
   begin real S; S:=0;                                          (5.28)
   for i:=1 step 1 until n do S:=S+y; SUM :=S
   end SUM;
```

Az (5.28) eljárás alapötlete hasonló, mint az (5.21) eljárásé, és lényegében ugyanarra használható. Alkalmazzuk például (5.28)-at ugyanannak a feladatnak a megoldására, amelyre (5.21)-et: két tízelemű vektor skaláris szorzatának meghatározására. A megoldást ott az (5.22) eljárásutasítás adta; teljesen egyenértékű utasítás a következő:

```
X:=SUM (k, 10, U[k]×V[k]);                                      (5.29)
```

(5.29)-et értékadó utasításként kellett megadnunk, mivel *SUM* függvényeljárás, tehát aritmetikai kifejezés. Tudjuk, hogy egy aritmetikai kifejezés értékével kiszámítás után mindig valaminek történnie kell: adott esetben ezt az értéket átadtuk az *X* változónak, amint az (5.22) esetében is történt, de ott a formális-aktuális paramétercsere közbejöttével.

Rendkívül gyakran fordul elő függvényeljárás, mint egy másik eljárás formális, ill. aktuális paramétere. Az alábbiakban két példát mutatunk erre.

Első példánk egy numerikus integrálási eljárás, mely tetszőleges $f(x)$ függvény $a$-tól $b$-ig vett integrálját számítja ki a Simpson-formula segítségével. [A Simpson-formulára nézve lásd (4.18)-at.] Maga az eljárásdeklaráció nem függvényeljárás formájában van felírva — bár ennek sem lenne semmi elvi akadálya —, viszont szerepel benne egy **real procedure**-ként specifikált *f* formális paraméter. Az eljárásdeklaráció:

```
procedure SPS (a, b, f, N, INT); value a, b, N;
real a, b, INT;
   integer N; real procedure f;                                 (5.30)
   begin real I, x, h; integer k; switch S:=S1, S2;
      k:=1; h:=(b-a)/N; I:=f(a)+f(b);
      for x:=a+h step h until b-h do begin go to S[k];
```

```
S1: I:=I+4×f(x);  k:=2;  go to L;
S2: I:=I+2×f(x);  k:=1;  L: end x;
INT:=h/3×I end SPS;
```

Az (5.30) eljárás fix lépésközzel integrálja az $f(x)$ függvényt az $[a, b]$ számközben, oly módon, hogy az integrációs intervallumot $N$ részre osztja fel. Maga az algoritmus rendkívül hasonlóan működik a (4.19) programhoz — úgy is mondhatjuk, hogy ugyanannak a feladatnak egy lehetséges másik megoldása — azonban egy igen lényeges különbség van a kettő között, és ez éppen az $f$ formális paraméterrel kapcsolatos: az (5.30) eljárás ui. bármely olyan függvény integrálására felhasználható, amelynek integrandusát egy ALGOL függvényeljárás-deklarációval le lehet írni. Tehát lényegesen általánosabb a (4.19) egyedi programnál, amely kizárólag annak a függvénynek az integrálására használható, amelyhez készült, és ha egy másik függvényt akarnánk vele integráltatni, akkor az integrandust generáló utasításokat ki kellene benne cserélni.

Tegyük fel, hogy ki akarjuk számíttatni az alábbi integrált:

$$I = \int_0^1 \frac{dx}{1+2x+4x^{3/2}}. \tag{5.31}$$

mégpedig oly módon, hogy a $[0, 1]$ intervallumot 16 részre osztjuk fel. E feladat megoldására a következő teljes program írható:

```
begin procedure SPS (a, b, f, N, INT);
   {az (5.30) eljárásdeklaráció folytatása}
   end SPS;

real II;  real procedure F(x);  value x;  real x;
   F:=1/(1+2×x+4×x↑1.5);  SPS (0, 1, F, 16, II);
   {Integrál értékének kinyomtatása} end program
```

Látjuk tehát, hogy egy konkrét függvény integrálásához nem kell egyebet tennünk, mint egy eljárásdeklarációt írnunk az integrandus generálására, és ennek azonosítójá-

val, mint aktuális paraméterrel, aktivizálnunk az integráló eljárást.

Következő példánkban egy eljárást mutatunk be közönséges differenciálegyenletek integrálására* Runge—Kutta-módszerrel. Legyen az elsőrendű explicit differenciálegyenlet

$$y' = f(x, y), \tag{5.32}$$

és tegyük fel, hogy ismeretes a megoldás valamely $x_1$ helyen; jelöljük ezt $y_1$-gyel. Meg akarjuk határozni a megoldás értékét az $x_2 = x_1 + h$ helyen, ahol $h$ egy adott lépéshossz. Ekkor $y_2$-t közelítőleg a következő formulák alapján számíthatjuk ki:

$$\begin{aligned} k_1 &= hf(x_1, y_1); \qquad (5.33)\\ k_2 &= hf(x_1 + 0.5h, y_1 + 0.5k_1);\\ k_3 &= hf(x_1 + 0.5h, y_1 + 0.5k_2);\\ k_4 &= hf(x_1 + h, y_1 + k_3);\\ y_2 &= y_1 + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4]. \end{aligned}$$

Az (5.33) formulák segítségével eljutunk a differenciálegyenlet megoldásához egy „következő" pontban, ha ismeretes a megoldás értéke az „előző" pontban. E* formulák $h$ előzetes ismeretét tételezik fel, azaz alkalmazásukat meg kell előznie a lépéshossz megválasztásának; ez történhet az adott konkrét feladat előzetes numerikus analízise alapján, vagy a számítás menetével egy időben, a konkrét számokból merített „tapasztalatok" alapján. Nyilvánvaló, hogy a menet közben való megválasztás időigényesebbé teszi a számítást, és ezért, ha van rá mód, célszerűbb a feladatot előzetesen numerikus vizsgálatok tárgyává tenni. Sajnos azonban a gyakorlati feladatok nagy részében az előzetes vizsgálat keresztülvihetetlen, mert azok a formulák, amelyekkel a lépéshossz becsülhető lenne, olyan mennyiségeket tartalmaznak, melyek nem állnak rendelke-

* A programot némi egyszerűsítéssel az ALGOL-jelentésből vettük át.

zésre, és még hozzávetőleges meghatározásuk is igen hosszadalmas, ha ugyan egyáltalában lehetséges. Mindezen okok következtében a gyakorlatilag használható differenciálegyenlet megoldó programoknak lehetőséget kell biztosítaniuk a lépéshossz automatikus megválasztására. (Megjegyezzük, hogy a numerikus integrálás esetén hasonló a helyzet, bár a lépéshossz megbecslése ott talán valamivel könnyebb.)

A következő program tehát automatikusan választja meg a lépéshosszt. Az algoritmus annyiban érdekes, hogy felfogható mint egészen primitív „tanuló" program, amely „okul" korábbi tapasztalataiból, ui. ezek későbbi viselkedését befolyásolják.

A választás elve a következő. Tegyük fel, hogy adva van a differenciálegyenlet megoldása valamely $x_1$ helyen, és ez egyenlő $y_1$-gyel. Meg akarjuk határozni a megoldást valamely $x_2$ helyen: $y(x_2)=y_2$. A numerikus analízis elméletéből ismert, hogy $x_1$-ből több rövidebb lépésben kiszámított függvényértéket használva, általában nagyobb pontossággal kapjuk $y_2$-t, mint kevesebb „közbülső" osztópont beiktatásával. Két különböző lépéshosszal végzett kísérlet eredményének összehasonlítása alapján következtetéseket lehet levonni a közelítések pontosságáról. Bár az elmélet óvatosságra int e következtetések jogosultságát illetően, a gyakorlati számításokban csaknem mindig ezekre támaszkodunk.

A bemutatandó ALGOL-eljárás a lépéshossz szukcesszív felezésén alapul. Először az eljárás megkísérli egy lépéssel „átlépni" az $[x_1, x_2]$ intervallumot, és hogy meggyőződjék a kapott pontosságról, kiszámítja fele lépéshosszal is a megoldás közelítő értékét. Ha azt tapasztalja, hogy a teljes és a fél lépéshosszal végzett kísérletek eredménye nem egyezik kellően, akkor a fél lépéshosszt tekintve egésznek, megkísérli kiszámítani a megoldást az $[x_1, x_2]$ intervallum felezőpontjában. Ha még most sem kielégítő az egyezés, ismét felezi a lépéshosszt, és megpróbálja kiszámítani a megoldást az intervallum negyedelő pontjában, s így tovább. Ha egyszer azt tapasztalta, hogy az intervallum hosszának $2^n$-ed részével mint lépéshosszal végzett kísérlet eredménye ele-

gendő pontossággal egyezik az intervallum $2^{n+1}$-ed részével végzett kétlépéses kísérlet eredményével, akkor a megoldást meghatározottnak tekinti az $x_1 + \frac{x_v - x_1}{2^n}$ pontban. A továbbiakban az e pontban vett megoldást tekinti kezdőértéknek, és megpróbál továbbjutni. A lépéshosszt most az előző kísérletek alapján adottnak tekinti, és mindjárt az $x_1 + \frac{x_v - x_1}{2^{n-1}}$ pontban kísérli meg meghatározni a megoldást, és csak akkor felezi tovább a lépéshosszt, ha azt találja, hogy a következő lépés véghezviteléhez az már nem kielégítő. Ha ezzel szemben sikerül továbblépnie változatlan lépéshosszal, akkor megjegyzi magának, hogy az adott lépéshosszal már két ízben sikeresen továbblépett. Az azonos lépéshosszal végzett egymásutáni sikeres kísérletek számát nyilvántartja, és ha ez eléri az ötöt, akkor — abból a meggondolásból, hogy hátha szükségtelenül kicsi a lépéshossz — megpróbálja duplázni, hogy gyorsabban jusson előre. Ez azért fontos, mert a gyakorlati feladatok nagy részében az igen rövid lépéshossz esetleg csak egy rövid szakaszon indokolt. Ha a duplázás nem jár sikerrel, akkor tartja magát az eredeti lépéshosszhoz további öt lépésig, és ezután ismét kísérletet tesz a duplázásra. Közben természetesen állandóan ellenőrzi, hogy nem kell-e felezni, és ha a pontosság lecsökken, azonnal megfelezi a lépéshosszt, és úgy halad tovább. Az eljárás tehát „kitapogatja" a megoldás kényes pontjait, és csak ott számol kicsiny lépésközzel, ahol feltétlenül szükséges. A programban külön vizsgálat gondoskodik arról, hogy az eljárás a végén nehogy „átlépje" az $x_v$ pontot. Erről azért kell külön gondoskodni, mert az algoritmus esetleg éppen olyan helyen duplázná meg a lépésközt, hogy a kétszeres lépésközzel kikerülne az $(x_1, x_v)$ intervallumból.

Az eljárásdeklaráció a következő:

```
procedure RK (x, y, FGV, eps, xV, yV, fi); value x, y, eps, fi;
   real x, y, xV, yV, eps; Boolean fi; real procedure FGV;
begin real x1, x2, x3, y1, y2, y3, H; own real Hs;
```

```
own integer s;  integer k, j;  Boolean ki;
procedure RK1L (x, y, h, xv, yv); value h;  real x, y, xv,
yv, h;
   begin real z, w;  integer j;  array a[1:5];
   a[1]:=a[2]:=a[5]:=h/2;  a[3]:=a[4]:=h;
   xv:=x;  yv:=w:=y;                                    (5.34)
   for j:=1 step 1 until 4 do
      begin z:=FGV (xv, w);  xv:=x+a[j];
      w:=y+a[j]×z;
      yv:=yv+a[j+1]×z/3 end j end RK1L;
comment Itt kezdődik a tulajdonképpeni eljárástörzs;
if fi then begin H:=xV-x;  s:=0 end else H:=Hs;
ki :=false;
AA : if (x+2.01×H-xV>0)≡(H>0) then
   begin Hs :=H;  ki :=true;  H:=(xV-x)/2 end;
   RK1L (x, y, 2×H, x1, y1);
BB: RK1L (x, y, H, x2, y2);  RK1L (x2, y2, H, x3, y3);
   if abs ((y1-y3)/y3)>eps then go to CC;  x:=x3;  if ki
   then go to DD;
   y:=y3;  if s=5 then begin s:=0;  H:=2×H end;
   s:=s+1;  go to AA;
CC: H:=0.5×H;  ki:=false;  x1:=x2;  y1:=y2;
s:=0;  go to BB;
DD: yV:=y end RK;
```

Szükséges néhány további megjegyzést tennünk az eljárás működésének teljes megvilágítására. Az eljárástörzs blokk, melynek fejében néhány munkarekesz jellegű változón kívül deklarálva van egy belső eljárás. Ez az *RK1L* azonosítót viseli, és szerepe az (5.33) formulák egyszeri kiszámítása, adott kezdőérték és lépésköz mellett. Tehát *RK1L* maga is „közérdekű" eljárás, és ha az automatikus lépésközválasztásra nincs szükség, akkor használható, mint „egylépéses" eljárás (innen kapta azonosítóját). *RK* eljárástörzse valójában vezérprogram, amely a lépéshossz rövidítését és hosszabbítását szabályozza. A lépéshosszal való manipulációkat eléggé részletesen leírtuk, és úgy hisszük, hogy az

Olvasó ennek alapján el tud igazodni az utasítások között. Szólnunk kell viszont az eljárás *fi* logikai típusú bemenő paraméteréről, valamint az *s* és *Hs* **own** típusú lokális változókról. Ezek hárman kapcsolatban vannak egymással, és azt célozzák, hogy az *RK* eljárás ismételt behívása esetén lehetővé tegyék az előzőkben szerzett tapasztalatok hasznosítását. Ennek érdekében az *s* változó a következő behívásig megőrzi az azonos lépéshosszal megtett lépések számát, *Hs* pedig ezt az azonos lépéshosszt tárolja. Ha az eljárás *fi* = **false** aktuális értékkel kerül behívásra, ez azt jelenti, hogy ismételt, nem első behívásról van szó, és lehetőség van a korábbi tapasztalatok hasznosítására. Ha a *fi* paraméter a behíváskor **true**-val egyenlő, akkor a „tapogatódzás" újonnan kezdődik, előzetes tapasztalatok felhasználása nélkül. A *fi* = **false** lehetőség akkor használható jól ki, ha például az eljárást a differenciálegyenlet megoldásának tabellázására használjuk, mert ilyenkor — némi kezdeti próbálkozás után — „beáll", és viszonylag kevés adminisztratív művelettel végzi a számolást.

Az (5.34) program igen illusztratív, az ALGOL-nak rendkívül sok lehetősége ki van benne használva. Többek között az **own** típusú változók gyakorlati alkalmazását látjuk benne, és mindjárt levonhatjuk azt a következtetést, hogy az **own** koncepciónak elsősorban eljárásokkal kapcsolatban van fontos szerepe, éppen olyan esetekben, amikor a program „emlékszik" a saját „előéletére", azaz valamit meg akar őrizni a korábban kiszámított mennyiségekből. Közönséges blokkok esetében ez mindig megoldható globális változók alkalmazásával, eljárások törzsében azonban — amelyek az eljárás aktivizálásánál beskatulyázott blokká válnak — globális azonosítók használata a már kifejtett okok miatt nem célszerű.

Az (5.30) és (5.34) eljárásdeklarációk alapján nyilván világos, hogy az eljárások függvényeljárás típusú paraméterei központi helyet foglalnak el az ALGOL-ban, mert minden olyan eljárásdeklarációnak, amely egy tetszőleges függvény értékein akar dolgozni, szükségképpen van függvényeljárásként specifikált formális paramétere. Ilyenek többek között az integráló, differenciálegyenlet-megoldó,

gyökkereső, szélsőérték-kereső stb. eljárások, tehát a numerikus analízis tipikus feladatainak legnagyobb része. Emiatt a függvényeljárások az ALGOL legfontosabb szintaktikus egységei közé tartoznak. Sajnos, van olyan gépi reprezentáns, amelyben eljárások formális paramétere nem lehet (sem függvény-, sem egyéb) eljárás; ez igen rontja az illető gép ALGOL-rendszerének hatékonyságát, annak ellenére, hogy kis ügyeskedéssel általában ez a megszorítás is kikerülhető.

Az alábbiakban a függvényeljárásoknak egy sajátságos tulajdonságára, az ún. *mellékhatások* (angolul: *side effect*) lehetőségére fogunk rámutatni. Egy függvényeljárásról akkor mondjuk, hogy mellékhatása van, ha aktivizálása közben megváltoztatja olyan változók értékeit, amelyek abban a blokkban, amelyben a függvényeljárás deklarálva van, érvényes deklarációval rendelkeznek. A mellékhatás létrejöhet oly módon, hogy az eljárástörzsben értékadás történik egy vagy több, az eljárástörzsre globális azonosítóra nézve, vagy oly módon is, hogy a függvényeljárásnak saját azonosítóján kívül is van kimenő formális paramétere, és az eljárás aktivizálásakor ez is értéket kap. Lássunk mindkettőre egy-egy példát!

Az első fajta mellékhatást produkálja például a következő eljárás:

```
real procedure SDFL (x); value x; real x;
begin y:= y+1; SDFL:= 1+x|2 end                    (5.35)
```

Feltesszük, hogy $y$ érvényes deklarációval rendelkezik abban a blokkban, amelyben (5.35) deklarálva van. (Legegyszerűbb esetben ugyanannak a blokknak lokális azonosítója.) Tekintsük e blokk törzsének a következő kiragadott részletét (valamennyi itt szereplő azonosítóról feltesszük, hogy deklarálva van):

```
p:= y:= 1; m1:= y+SDFL (p);
y:= 1; m2:= SDFL (p)+y; {folytatás}                (5.36)
```

Az (5.36) példában két teljesen azonos kezdeti körülmények között végrehajtott értékadó utasítás található; a kettő

semmi másban nem különbözik, mint az összeadandók sorrendjében, a változók aktuális értékei az utasítás végrehajtásának kezdetekor megegyezők. Mégis a programrészlet végén $m1$ és $m2$ értéke különböző ($m1=3$ és $m2=4$), mivel az *SDFL* eljárás aktivizálása közben $y$ értéke megváltozik, és a balról jobbra szabály értelmében (lásd II. fejezet 5. p.), az első értékadásnál, ahol $y$ áll elöl, $y$ korábbi értéke kerül felhasználásra, míg a második értékadásnál, ahol $y$ hátul áll, már $y$ módosított értékével kell dolgozni.

Mint mondottuk, létrejöhet mellékhatás anélkül is, hogy az eljárástörzsnek globális azonosítója volna. Tekintsük például a következő eljárásdeklarációt:

```
real procedure SDFF (x, z); value x; real x, z;
begin z := z + 1; SDFF := 1 + x↑2 end
```
(5.37)

Az alábbi program (5.36)-tal teljesen azonos effektust produkál:

```
p := y := 1; m1 := y + SDFF (p, y);
y := 1;
m2 := SDFF (p, y) + y; {folytatás}
```

A mellékhatások lehetőségével minden ALGOL fordítóprogram szerkesztésekor számolni kell. Éppen emiatt van egy sereg szintaktikus és szemantikai szabály, amelynek első pillanatban nem látszik az értelme. Ezek közé tartozik — hogy csak a legfontosabbakat említsük — a szigorú balról jobbra szabály, amelyet az esetleg kényelmesebb vagy gazdaságosabb kódolás érdekében sem szabad megszegni; az értékadó utasítások szemantikájának az a szabálya, hogy a bal oldali változók indexkifejezései előbb számítandók ki, mint a jobb oldali kifejezés aktuális értéke, és magukra az indexkifejezések kiértékelésére is a balról jobbra szabály érvényes (mind az indexkifejezéseknek, mind a jobb oldali kifejezéseknek a kiértékelése produkálhat mellékhatást); a logikai kifejezések tárgyalása során kimondott azon szabály, mely szerint a kifejezés kiszámításában akkor sem szabad megállni, ha valamely részeredmény már

eleve megszabja a végeredményt (pl. logikai szorzat első tényezője **false** stb.).

Sajnos, a mellékhatások lehetősége rendkívül megnehezíti az ALGOL fordítóprogramok által produkált célprogramok optimalizálását. A FORTRAN nyelv szerzői nem vállalják a felelősséget esetleges mellékhatások fellépéséért, és figyelmeztetik a programozót, hogy a fordítóprogram fenntartja magának a jogot az elemi algebra azonosságainak alkalmazására fordítás közben (kommutatív, asszociatív, disztributív törvény). Tehát a FORTRAN-programokban a programozónak kell ügyelnie arra, hogy ne írjon olyan programot, amelynek végrehajtásában nemkívánatos hatások lépnek fel. Ez a megkötés lehetővé teszi viszont a célprogramok optimalizálását, ismétlődő részkifejezések felismerését, és az ismételt számolás elkerülését, indexkifejezéseknek cikluson kívüli számolását stb. Mindez az ALGOL-ban (egyébként nem kizárólag a mellékhatások lehetősége miatt) nagyon nehézkes, ami kétségkívül egyik oka annak, hogy egyesek idegenkednek az ALGOL gyakorlati alkalmazásától. A mellékhatások kérdésére az ALGOL-bizottság még vissza szándékozik térni.

## 5. Rekurzív eljárások és rekurzív eljárásbehívások

Láttuk az előző fejezetekben, hogy az ALGOL-eljárások egymást is aktivizálhatják, mégpedig akár az eljárás végrehajtása közben, akár az eljárás aktuális paramétereinek kiértékelése közben. Az (5.30) eljárásdeklaráció például eljárástörzsének végrehajtása közben aktivizálja az *f* eljárást, amely formális paramétere. Egy konkrét függvény integrálásakor az *f* eljárás helyén megadott aktuális függvényeljárás kerül aktivizálásra, minden olyan helyen, ahol (5.30) eljárástörzsében *f*-re hivatkozás van. Az *f*-nek megfelelő aktuális eljárásparaméter vagy ugyanabban, vagy egy magasabb rendű blokkban van deklarálva, mint maga az eljárás. Az (5.34) eljárás is aktivizál végrehajtása közben másik eljárást, *RK1L*-et, amely az eljárástörzs blokkfejében van deklarálva, ez pedig ismét egy harmadikat: *FGV*-t.

Az is megengedett, hogy egy eljárás törzse az eljárástörzsre nézve globális eljárásokat aktivizáljon. Mindenesetre látszik, hogy egy eljárás működése közben igen sokféle módon kerülhetnek aktivizálásra más eljárások, sok esetben aktivizálások egész láncolata jön létre.

Az ALGOL szintaktikus és szemantikai szabályai azt is megengedik, hogy egy eljárás önmagát hívja be. Ilyen esetben azt mondjuk, hogy az eljárás *rekurzíve aktivizálódott*. A rekurzív aktivizálásnak két alapesetét különböztetjük meg, mindkettőn belül további 2−2 aleset lehetséges. A két alapeset a következő:

1. Valamely eljárásutasításban az aktuális paraméterek listáján az aktivizálandó eljárás azonosítója szerepel;

2. valamely eljárás az eljárástörzs végrehajtása közben — közvetlenül vagy közvetve — önmagát hívja be.

Az első esetben *rekurzív eljárásbehívásról*, a másodikban vagy *rekurzív eljárásról*, vagy *szimultán rekurzív eljárásrendszerről* beszélünk, aszerint, hogy az eljárás közvetlenül vagy közvetve hívja-e be önmagát.

Foglalkozzunk először a rekurzív behívással! A rekurzív behívás első és egyszerűbb alapesetével akkor állunk szemben, ha az eljárás valamelyik érték szerint behívott formális paraméterére olyan aktuális paraméterkifejezés van megadva, amelynek kiszámításához magának a függvényeljárásnak az aktivizálása szükséges. Tekintsük például az (5.27) eljárásdeklarációnak megfelelő *sh* eljárást, és tegyük fel, hogy a programban a következő értékadó utasítást találjuk:

$$x := sh(1 + y \times sh(t)); \qquad (5.38)$$

[feltesszük, hogy az (5.38) utasításban valamennyi változó deklarált, és valós típusú]. Az (5.38) utasítás az *sh* eljárást aktivizálja, és az aktuális paraméterkifejezésben is szerepel *az sh* azonosító. Ez a rekurzív aktivizálás legegyszerűbb esete, melyet rendszerint még erősen korlátozott gépi reprezentánsok is megengednek, értelmezése magától értetődik, így részletes elemzése fölösleges.

A rekurzív eljárásbehívás második alapesete áll fenn, ha egy eljárás név szerint behívott formális paraméteré-

nek helyén aktuális paraméterként saját magát az eljárást adjuk meg. Az ilyen rekurzív eljárásbehívásnak illusztrálására írjunk ALGOL-programot egy $n \times n$ típusú négyzetes mátrix elemei négyzetösszegének kiszámítására. Rendelkezésre áll az (5.28) eljárás. A kiszámított négyzetösszeg értékét az *NSG* nevű változó vegye fel. A megfelelő hatás elérhető egyetlen értékadó utasítás segítségével:

$$NSG := SUM\ (i, n, SUM\ (k, n, a[i, k]\uparrow 2)); \qquad (5.39)$$

A *SUM* eljárásnak *y* név szerint behívott formális paramétere, és mivel az eljárástörzsben *y* nem szerepel értékadó utasítás bal oldalán, az eljárás aktivizálásakor tetszőleges aritmetikai kifejezés megadható aktuális paraméterként. A *SUM* függvényeljárás aktuális argumentumaival együtt korrekt aritmetikai kifejezés, ezért szintaktikusan kifogástalan ALGOL-programot kapunk, ha *SUM* harmadik formális paraméterének helyén egy magától *SUM*-tól függő aritmetikai kifejezést adunk meg aktuális paraméterként. Ezt tettük az (5.39) értékadó utasításban, és ezzel a fogással egyetlen értékadó utasítással tudtuk leírni azt, amit egyébként „hagyományos" módszerek alapján egy kétszeresen egymásba skatulyázott **for** ciklussal kellett volna leírnunk. A bemutatotthoz hasonló programozási fogások aránylag gyakran alkalmazhatók a gyakorlatban, és a fordítóprogramok többsége megengedi ezt a fajta rekurzív behívást is. Rekurzív behívás segítségével felhasználhatunk pl. egy egyszeres integrált kiszámító eljárást kétszeres integrálok kiszámítására, egyváltozós függvény szélsőértékét megkereső eljárást kétváltozós függvény szélsőértékének megkeresésére stb. A rekurzív behívás természetesen akárhányszoros mélységben megengedett, tehát pl. (5.28) segítségével akárhányszoros szummák előállíthatók.

Az (5.38) és az (5.39) típusú rekurzív hívás között látszólag nincs jelentős elvi különbség, azonban attól függően, hogy az eljárás azonosítója név vagy érték szerint behívott formális paraméter helyén fordul-e elő, más és más a fordítóprogram teendője; a név szerinti hívás esetén a fordítás bonyolultabb. Az alkalmazott fordítási technika különbözősége indokolja a két aleset megkülönböztetését.

Áttérünk a rekurzív eljárások tárgyalására. A rekurzív eljárások fogalmát az irodalomban — szinte hagyományosan — a faktoriális függvény kiszámításának ALGOL-programján szokták megmagyarázni.

A faktoriális függvényre a következő rekurzív definíció adható meg:

$$n! = \begin{cases} 1, & \text{ha} \quad n=0 \\ n \times (n-1)!, & \text{ha} \quad n>0 \end{cases} \tag{5.40}$$

($n$ pozitív egész szám). Ez szinte változtatás nélkül átírható ALGOL-programmá:

```
integer procedure FAC (n); value n; integer n;
FAC:=if n=0 then 1 else FAC (n-1)×n;
```
(5.41)

Formálisan könnyű látni, hogy az (5.41) eljárásdeklaráció valóban az (5.40) definíció ALGOL átírása, azonban fel szokott merülni a kérdés, hogy valójában mi történik a gépben egy ilyen eljárás aktivizálása során, hogyan „dolgozik" egy rekurzív eljárás?

A kérdésre megpróbálunk egy olyan példa alapján választ adni, amely nem az ALGOL-terminológián alapul, hanem teljesen hétköznapi fogalmakkal dolgozik.

Képzeljünk el egy sokemeletes iroda-toronyházat, amelynek minden emelete azonos beosztású, minden emeleten van egy sereg hivatalszoba, és minden szobában ül egy hivatalnok, aki utasításokat ad a hozzá forduló ügyfeleknek. Az utasítások a következők lehetnek:

1. „Menjen ugyanezen az emeleten a ... számú szobába!"

2. „Menjen az emelet kijáratához!"

3. „Menjen a fölöttünk levő emelet első szobájába, és ha a fölöttünk levő emeleten végzett, jöjjön ide vissza!"

4. Ezenkívül a hivatalnok intézhet valamit az ügyfél iratain, és utasítás nélkül továbbengedheti, amely esetben az automatikusan a következő szobában ülő hivatalnokhoz megy.

A hivatalnokoknak nincs joguk az ügyfelet sem alacsonyabb emeleten fekvő szobába küldeni, sem pedig olyanba, amelyik két vagy annál több emelettel van magasabban; sőt az eggyel magasabb emeleten fekvő szobák közül is mindig csak az elsőbe küldhetik, és az onnan megy majd esetleg az emelet további szobáiba.

A 2. számú utasítás hatására, ha egy földszinten ülő hivatalnok adta, az ügyfél befejezte látogatását a hivatalban és távozik; ügye elintézettnek tekintendő. Ha ezzel szemben ugyanezt az utasítást egy emeleti hivatalszobában kapja, akkor ez azt jelenti, hogy visszatér az eggyel alacsonyabb számú emeletre ahhoz a hivatalnokhoz, aki legutoljára küldte egy emelettel feljebb, és ott újabb utasítást kap.

Az ügyfél célja, hogy véges idő alatt elintézze ügyét a hivatalban. Ha ügye olyan, hogy az a hivatal fenti rendtartása mellett elintézhető, akkor véges sok szoba felkeresése után eljut egy olyan földszinti szobához, amelyben elhangzik a „Menjen az emelet kijáratához" utasítás, és ez véget vet a szobák közti bolyongásnak. Ha ellenben az ügy nem intézhető el az adott rendtartás mellett, akkor az ügyfél a végtelenségig bolyong a szobák között, és soha többé nem jön ki a hivatal kapuján. A legegyszerűbb ügyek a földszinten intéződnek, és ilyenkor az ügyfélnek nem kell lépcsőt járnia. Bonyolult ügyek esetén az ügyfelet magas emeletekre is felküldik az ügyintézés során.

A közönséges, nem rekurzív eljárás olyan „hivatal", ahol mindent elintéznek a földszinten. Rekurzív eljárás esetén lehetőség van a vezérlést arra utasítani, hogy eggyel magasabb szinten járja végig mégegyszer a program törzsét; itt esetleg ismét még eggyel magasabb szintre kerül a vezérlés, és így tovább. Rekurzív eljárás végrehajtása közben a gép nyilvántartja, hogy éppen hányadik szinten tartózkodik, és az eljárástörzset bezáró **end** elérése csak akkor jelenti a számára a főprogramra való visszatérést, ha egyidejűleg a szintszámláló nullán áll. Ellenkező esetben az **end** elérése után a gép tudja, hogy nem az egész eljárással végzett, hanem annak csupán egy szintjével, megfelelően az előző példánkban annak az esetnek, amikor az ügyfél

nem a földszinten kap „Menjen a kijárathoz" utasítást. Az ALGOL megengedi az eljárásoknak tetszőleges magas szinten való aktivizálását, de a rekurzív hívásnak értelmesnek kell lennie, azaz lehetővé kell tennie a gép számára, hogy véges sok lépésben kijusson az eljárásból, ellenkező esetben kijárat nélküli, végtelen rekurzió lép fel. Ez történik pl. abban az esetben, ha az (5.41) algoritmust negatív argumentummal hívják be.

A rekurzív eljárásokkal kapcsolatban számos szintaktikus és szemantikai probléma merül fel, de ezekre e könyv keretében nem térünk ki. Rekurzív eljárásokat ugyanis az ALGOL tipikus alkalmazásaiban nem használunk.

A numerikus analízis legtöbb számítási algoritmusa ugyan rekurzív módon van adva, a gyakorlatban azonban a rekurzív algoritmust ciklikus programmá írjuk át. Meszsze vezetne annak taglalása, hogy ez mikor lehetséges, de a numerikus analízis és az ALGOL egyéb tipikus felhasználási területein általában sikerül, és ilyenkor a kényelmesebb programozás és a gyorsabb számolási idő érdekében ciklusos programmal és nem rekurzív eljárással számolunk. Például a faktoriális függvényt is célszerűbb az (5.41) algoritmus helyett egy közönséges **for** ciklus segítségével előállítani:

```
integer procedure FACD (n); value n; integer n;
begin integer i; real f;
f := 1; for i := 1 step 1 until n do f := f × i;
FACD := f end FACD
```
(5.42)

Az (5.42) eljárás a számológép-programozás gondolatköréhez szokott szakember számára sokkal természetesebb, mint a rekurzív felfogás, és nyilván a faktoriális kiszámítását mindenki így, és nem rekurzív eljárással programozná. A programozó a rekurzív formulát — ha tudja — mindjárt fejben átfogalmazza ciklikus algoritmussá. Azonban van olyan eset, amikor az egyedüli lehetőség rekurzív algoritmusok alkalmazása. Ilyen például a különböző fordítóprogramok megírása, amelyben kiterjedten alkalmaznak

rekurzív programokat.* Csakhogy ezek általában éppen nem ALGOL-ban íródnak, ezért az ALGOL-on belül ritkán merül fel rekurzív eljárások alkalmazásának szükségessége.

Hasonlóan ritkán, vagy még ritkábban merül fel a rekurzív aktivizálás negyedik alapesetének, az ún. szimultán rekurzív eljárásrendszerek alkalmazásának szükségessége. Szimultán rekurzív eljárásrendszerről akkor beszélünk, ha két eljárás — mondjuk *A* és *B* — kölcsönösen behívják egymást; pl. *A* aktivizálása esetén behívja *B*-t, majd *B* „visszahívja" *A*-t, és ez ismét *B*-t, s így tovább. Előbbi példánkat könnyű átfogalmazni erre az esetre — vagy akár *N* számú eljárásból álló szimultán rekurzív eljárásrendszer esetére — is.

Annak érdekében, hogy a különböző azonosító típusok között valamelyes rendet teremtsünk, összefoglaljuk, hogy milyenfajta azonosítók fordulhatnak elő az eljárások törzsében.

1. *Lokális azonosítók*: semmiféle kapcsolatuk nincs az eljárást aktivizáló programmal, érvényük, hatáskörük csak az eljárástörzsre terjed ki, az eljáráson kívülről semmilyen módon nem hozzáférhetők.

2. *Érték szerint hívott formális paraméterek*. Sok tekintetben hasonlóan viselkednek az eljárástörzs lokális azonosítóihoz, de definíciószerűen az eljárástörzset körülvevő azon fiktív blokkban tekintendők lokálisnak, amely magában foglalja a reájuk vonatkozó ugyancsak fiktív értékadó utasításokat is (lásd 3. p.). Következésképpen egyirányú kapcsolatban állnak az eljárást aktivizáló programmal: ez megadja kezdeti értéküket, de ezután függetlenné válnak a behívó programtól, és teljesen úgy viselkednek, mint a lokális változók. Az eljárástörzs végrehajtása után végértékük hozzáférhetetlen a behívó program számára.

3. *Név szerint hívott formális paraméterek*. Minden olyan helyen, ahol az eljárástörzsben előfordulnak, az eljárás minden egyes aktivizálásakor átadják helyüket a nevükben működő aktuális paramétereknek. Feladatuk a behívó programmal való kétoldalú kapcsolat fenntartása. Működésü-

* Lásd például [22].

ket az eljárástörzs „kitöltési" szabálya határozza meg (lásd 2. p.).

4. *Globális azonosítók.* A behívó programmal való kétoldalú kapcsolat fenntartása céljából alkalmazhatunk az eljárástörzsre nézve globális változókat is, amelyek lehetnek akár globálisak, akár lokálisak arra a blokkra nézve, amelyben az eljárás maga deklarálva van. Függvényeljárások esetében azonban globális változókra való értékadás az eljárástörzs végrehajtása közben nehezen áttekinthető mellékhatásokra vezethet (lásd 4. p.), és „közérdekű" eljárásokban globális azonosítók alkalmazása nem célszerű (lásd 3. p.).

5. *Az eljárás saját azonosítója:*

*a)* Függvényeljárás esetében az eljárás saját azonosítójának legalább egyszer kötelezően elő kell fordulnia egy értékadó utasítás bal oldalán aktuális paraméterlista nélkül. Az ilyen értékadó utasítás a függvényeljárás segítségével meghatározandó függvényérték definiálására szolgál; a keletkező függvényérték típusát az eljárásdeklarációt bevezető típusdeklaráló alapjel (**real**, **integer**, **Boolean**) határozza meg (lásd 4. p.).

*b)* Az *a)* alatti esetet kivéve, minden olyan helyen, ahol az eljárástörzsben előfordul az eljárás azonosítója, az eljárás rekurzíve aktivizálódik. Magától értetődik, hogy rekurzív aktivizálás esetén az eljárás azonosítójának aktuális paraméterlistával együtt kell állnia, és az aktuális paraméterlistának ki kell elégítenie az eljárások behívásával kapcsolatos ismert szabályokat.

## 6. Paraméterelválasztók és idézetek

Ebben a fejezetben az ALGOL 60 hivatkozási nyelv két utolsó — a nyelv szempontjából kisebb jelentőségű — szintaktikus egységét ismerjük meg: a paraméterelválasztókat és az idézeteket.

Az eddigiekben az eljárások formális és aktuális paramétereinek elválasztására mindig a vesszőt alapjelet használtuk. Az ALGOL nyelv azonban megengedi, hogy ehelyett

*paraméterelválasztó jelsorozatot* használjunk; ez lehetővé teszi, hogy a formális (aktuális) paraméter mellé a programozó odaírjon valamilyen emlékeztető szöveget, amely felvilágosítást tartalmaz az illető paraméter jellegére vonatkozóan. Ezek a felvilágosítások a gép számára lényegében kommentárt jelentenek, amelyeket a fordítás során nem vesz figyelembe.

A paraméterelválasztó jelsorozatok formálisan a következő alakúak lehetnek:

)szöveg:( (5.43)

Pontosabban megfogalmazva, a paraméterelválasztó jelsorozat mindig egy végzárójellel kezdődik, utána tetszőleges szöveg (betűket és szóközöket tartalmazó jelsorozat) áll, ezt követi egy kettőspont, és az egész konstrukciót egy kezdőzárójel zárja be.

Az (5.43) típusú jelsorozatok használatát a következő példán illusztrálhatjuk:

**procedure** *SPS* (*a*, *b*) *integrandus* : (*f*) *kezdeti osztó pontszám* : (*N*) *integrálás eredménye* : (*INT*);

Példánkban az (5.30) eljárásdeklaráció bevezető sorát írtuk át oly módon, hogy a vesszők helyett néhány helyen elválasztó jelsorozatot alkalmaztunk. Az átírás semmit sem változtat az eljárásdeklaráció kezdetének eredeti szemantikáján; a paraméterelválasztó jelsorozat teljes mértékben egyenértékű a vesszővel, és alkalmazása megengedett, de semmilyen tekintetben sem kötelező (hasonlóan a kommentárokhoz). Hasonló paraméterelválasztók alkalmazhatók eljárásutasításokban is, aktuális paraméterek elválasztására. Alkalmazásuk itt is tetszőleges. Nincs olyan jellegű szabály, amely megkövetelné, hogy az eljárásdeklarációban és az eljárásutasításban alkalmazott esetleges paraméterelválasztás között bármilyen kapcsolat legyen. Ez azt jelenti, hogy a programozónak joga van pl. vesszőt használni az eljárásdeklarációban elválasztóként, és ugyanazon eljárás aktivizálásakor szöveget iktatni a paraméterek közé. Többszöri aktivizálás esetén a programozó ott és annyiszor

alkalmazza a paraméterek közötti kommentárt, ahol és ahányszor jónak látja, nincsen „mindenütt, vagy seholsem" jellegű szabály sem.

A legtöbb ALGOL gépi reprezentációban vannak olyan standard eljárások, amelyek tetszőleges jelsorozatokon, ún. *idézeteken* operálnak. Ezek nyilván olyan programok, melyeknek eljárástörzse nem ALGOL kódban van írva, mert, mint tudjuk, ALGOL-program nem tud számoktól és logikai értékektől különböző objektumokon dolgozni. Azonban a nem hivatkozási nyelven írt programok dolgozhatnak szövegekkel, jelsorozatokkal, és ezért a hivatkozási nyelv lehetőséget biztosít szövegeket tartalmazó eljárásutasítások felírására. Erre a célra az ALGOL-ban be van vezetve az idézet nevű szintaktikus egység. Formális szempontból, az idézet ALGOL alapjelek tetszőleges sorozata, egy kezdő és egy záró idézőjel közé zárva. Az idézetek tetszőleges mélységben egymásba skatulyázhatók, tehát bármely idézet tartalmazhat belsejében újabb idézeteket. A kezdő idézőjel (‘) és a záró idézőjel (’) ALGOL alapjelek.

*Példák*

‘Talpra magyar’
‘Egymásba skatulyázott ‘idézetek”

A kezdő és záró idézőjelekkel kapcsolatban az aritmetikai zárójelekre emlékeztető formális szabályok érvényesek: minden megkezdett idézőjelet be kell zárni, és bezárt idézőjelet előzőleg meg kell nyitni.

Tudjuk, hogy a hivatkozási ALGOL-ban a szóközöknek és sorváltásoknak semmi szerepük nincs (lásd II. fejezet 3. p.), tetszés szerint alkalmazhatók a jó tagolás és az áttekinthetőség érdekében. Ezzel szemben idézetekben a szóközöknek és a sorváltásoknak meghatározott jelentésük lehet, és ezért a szóköz jelölésére az ALGOL-ban külön alapjel van, a ␣, melyet kizárólag idézetekben használnak fel. Sorváltási jelet az ALGOL nem vezet be, de ALGOL-programok perforálásakor a bemenő lyukszalagon általában külön lyukkombináció felel meg a sorváltásnak. A szóközök és sorváltások nyilvántartása idézetek esetén azért fontos, mert az idézeteken éppen egy nem ALGOL-program

operál, és lehet, hogy abban a nyelvben, amelyen az idézeten operáló eljárás törzse íródott, jelentése van a szóköznek és a sorváltásnak is.

Mivel az idézetek már nem annyira a hivatkozási nyelv, mint a gépi reprezentációs nyelvek területére tartozó szintaktikus egységek, velük a hivatkozási nyelv keretében nem foglalkozhatunk, és használatukra az ALGOL-jelentés nem is ad szabályokat. A ki- és beviteli eljárások között általában találunk olyanokat, amelyek idézeteken operálnak, ilyenek pl. a szöveget kinyomtató eljárások.

Megjegyezzük, hogy egyetlen kivételes esetben lehet egy eljárás formális paramétere idézetként specifikálva akkor is, ha az eljárás maga ALGOL-ban van írva: amikor az eljárás behív egy nem ALGOL-ban írt eljárást, és az idézetparamétert csakis arra használja, hogy „továbbadja" aktuális paraméterül a behívandó nem ALGOL-eljárásnak. Idézet jellegű formális paraméter specifikálására a **string*** alapjel használható.

## 7. A periferiális egységekre hivatkozó utasítások

Az előző fejezetekben többször hangsúlyoztuk, hogy a gyakorlatban egy teljes ALGOL-program tartalmaz olyan utasításokat, amelyek végrehajtása közben a gép kiinduló információkat vesz fel a bemeneti egységen keresztül, és olyanokat, amelyek hatására közli a programozóval a számítás eredményeit. A gép a külvilággal a be- és kimeneti egységeken keresztül tartja a kapcsolatot; ezeket közös néven *periferiális egységeknek* nevezzük. A periferiális egység fogalomkörbe azonban nemcsak a nyomtató és beolvasó egységek tartoznak, hanem az összes olyan egység is, amely közvetlenül nem vesz részt a számításban, hanem csak adatokat tárol, vesz fel, vagy közöl. Nem tartozik a periferiális egységek közé a központi memória, mert a gép működése közben közvetlenül részt vesz a számításban, ezzel szemben a különböző segédmemóriákat (mágnesszalag,

* **string** = szimbólumlánc, jelsorozat.

mágnesdob, tárcsa stb.) már ide számítjuk. A géptípusok perifériális egységei messzemenően különböznek egymástól, sőt újabban azonos géptípuson belül is a megrendelő dönti el, milyen perifériális egységeket kíván vásárolni. Ezek kiválasztásakor igen alaposan fel kell mérnie, milyen célra kívánják a felállítandó gépet használni, mert a hatékony kihasználás úgyszólván ezen áll vagy bukik.

A perifériális egységek terén mutatkozó nagy eltérések miatt az ALGOL 60 szerzői úgy döntöttek, hogy nem definiálnak szabványos utasításokat a ki- és bevitel, valamint az egyéb perifériákra való hivatkozás céljaira. Minden felhasználónak magának kell kialakítania a számára legmegfelelőbb rendszert, természetesen oly módon, hogy az lehetőleg összeférjen az ALGOL elveivel és szabályaival.

Az ALGOL nyelv hivatalos definiálása óta eltelt hat év alatt igen nagyszámú — részben többé-kevésbé teljes, részben korlátozott — ALGOL gépi reprezentáns keletkezett, ami azzal járt, hogy nagymértékben megszaporodott a merőben különböző ki- és beviteli rendszerek* száma. Ez jelenleg igen erősen akadályozza az ALGOL általános felhasználását, mely ebben a tekintetben hátrányban van pl. a FORTRAN-nal szemben, amelynek lényegében egységes ki- és beviteli rendszere van.

Az elmúlt évek során tapasztalható fejlődés szemmel láthatóan arra mutat, hogy az ALGOL nyelvet előbb-utóbb ki fogják egészíteni egy szabványos ki- és beviteli rendszerrel. Lásd pl. [17], [18]. Az alábbiakban csupán arra szorítkozunk, hogy megkísérlünk áttekintést adni azokról az operációkról, amelyek egy gép ki- és beviteli rendszeréhez tartoznak.

A korszerű elektronikus számológépek programozási rendszereiben a perifériákkal kapcsolatos operációkat két nagy csoportba sorolhatjuk. Az elsőbe tartoznak azok az utasítások, amelyek az effektív adatcserét végeztetik a géppel (kinyomtatás, beolvasás, mágnesszalagos operációk

* A továbbiakban valahányszor ki- és beviteli rendszerről beszélünk, ezen mindig az összes szóbajövő periferiális egységre hivatkozó utasítások összességét értjük (tehát pl. a segédmemóriákkal való adatcsere műveleteit is).

stb.), míg a második csoport utasításai nem hajtanak végre semmit, csupán az adatcserében részt vevő jelsorozatok (nem okvetlenül számok!) formájának, hosszának, jellegének meghatározására, definiálására szolgálnak. Tegyük fel például, hogy ki akarjuk nyomtatni valamely programban az $x$ változó aktuális értékét. Legyen ez az érték mondjuk 7.5. Tegyük fel, hogy a „végrehajtó" utasítás a *print*; a *print* ($x$) utasítás hatására a gép kinyomtatja az $x$ változó aktuális értékét. A 7.5 számnak igen sokféle decimális ábrázolása lehetséges, és nagyon jó, ha a programozó meg tudja választani, hogy ezek közül melyik jelenjen meg az eredménylapon. (Kevésbé korszerű gépeken éppen az a baj, hogy csak egyféle lehetséges formátum létezik, és ez akadályozza áttekinthető eredménytáblázatok készítését.) E célból a programozó mellékel a nyomtatási utasításhoz egy formátumspecifikációt, amely megmondja, hogy az $x$ változó értéke milyen módon kerüljön kinyomtatásra: ez tartalmazza a kinyomtatandó jegyek számát összesen, esetleg a tizedespont utáni jegyek számát külön, és szükség esetén további rendelkezéseket a kinyomtatás módjáról. Egy lehetséges specifikáció pl. a következő:

$$format\ +d.ddd \tag{5.44}$$

melyet az $x$ szám kinyomtatására alkalmazva a következő nyomtatott alakot kapjuk:

$$+7.500 \tag{5.45}$$

Az (5.44) specifikáció jelentése ui. az, hogy a tizedespont előtt egy, utána három jegy nyomtatandó, és a szám elé kinyomtatandó az előjel is. Magától értetődik, hogy a kinyomtatandó számnak meg kell felelnie a specifikáció által támasztott hallgatólagos követelményeknek; (5.44) alapján pl. a 17.5 szám nyilván nem nyomtatható, a 0.0000175 szám pedig +0.000 alakban nyomtatódik ki. A ki- és beviteli rendszerek mindig biztosítják a decimális normálalakban való nyomtatás lehetőségét, amikor a szám egy decimális normáló tényezővel együtt jelenik meg a nyomtatott lapon. Egy ilyen lehetséges specifikáció:

$$format\ +.dddd+dd \tag{5.46}$$

Az (5.46) formátumban a 7.5 szám a következő alakban nyomtatódik:

$$+.7500+01 \qquad (5.47.1)$$

a 0.0000175 szám pedig így:

$$+.17500-04 \qquad (5.47.2)$$

Lehetségesek egyéb formátumok is; pl. adott értékes jegyszám mellett a gép oda teszi a tizedespontot, ahová a szám nagyságrendje szerint annak esnie kell. A

*format* +*ddddddddd* (5.48)

specifikáció mellett pl. a 7.5 szám

$$+7.5000000 \qquad (5.49.1)$$

alakban, a 0.0000175 szám pedig

$$+.00001750 \qquad (5.49.2)$$

alakban nyomtatódik.

Korszerű kinyomtató egységgel rendelkező elektronikus számológépeken egy sorban több szám is nyomtatható. Ilyen esetben a formátumspecifikációk nemcsak az egyes számok formáját határozzák meg, mint azt az (5.44) — (5.49.2) példákon láttuk, hanem az egy sorban álló számok elválasztásának módját is, pl. a közöttük hagyandó üres helyek számát, vagy azokat a jeleket, amelyek a számok közé elválasztó jelsorozatként nyomtatódnak. Elválasztó jelsorozat lehet például egy vessző és néhány üres hely, vagy akár egy magyarázó szöveg, amely felvilágosít a kinyomtatott szám jelentéséről.

Ahhoz, hogy egy számítás eredményét jól áttekinthető formában kapjuk, és hogy az ügyeletes kezelő a számítás pillanatnyi helyzetét figyelemmel kísérhesse, szükséges, hogy a gép képes legyen szövegeket kinyomtatni; ezek szolgálhatnak például egy számtáblázat fejléceként, vagy riasztó jelzést adhatnak valamely előre nem látott helyzet előállтáról. Alfanumerikus kivitel nélküli gépekre a matematikus — amikor egy feladatot átad futtatásra — kény-

telen részletes leírást (kezelési utasítást) mellékelni a programhoz, amely minden előreláthatő helyzetre felsorolja a kezelő teendőjét. A tapasztalat azonban az, hogy ha a kezelési utasítás bonyolult, akkor a kezelő vagy nem olvassa el, vagy elolvassa, de félreérti, és éppen nem azt csinálja, amit kellene. A szövegnyomtatás lehetőséget ad a program olyan kiképzésére, hogy minden kezelői beavatkozást a gép ír ki az adott pillanatban, és a kezelőnek csak az íróberendezést kell figyelnie, és aszerint eljárnia. Szövegek kinyomtatása esetén a formátumspecifikáció a szöveg hosszát (a kinyomtatandó jelek számát) tartalmazhatja.

Ugyancsak az áttekinthetőséget fokozza a nagyméretű számtáblázatoknak „lapokra" tördelése. Erről azt kell tudnunk, hogy a számológépek kiíróberendezései általában igen hosszú papírtekercsekről göngyölítik le a működtetésükhöz szükséges papírt; a gyakorlati felhasználhatóság szempontjából viszont célszerű, ha egy többméteres papírkígyó helyett több kisebb, ennélfogva jobban kezelhető résztáblázatot nyomtatunk ki. A gép ezért megszámlálhatja a kinyomtatott sorokat, és adott számú (pl. 25, 32, 40 stb.) sor kinyomtatása után nagyobb hézagot hagy, esetleg a következő sor előtt lapszámot is nyomtat. A számítás befejezése után az eredménypapírt szétvághatjuk, és pl. dossziéba fűzve, kitűnően kezelhető táblázatot kapunk. A kinyomtatási formátum meghatározásához hozzátartozik az egy lapra nyomtatandó sorok, valamint az egy sorban nyomtatandó jelek számának meghatározása is. (Utóbbit általában a kinyomtató egység műszaki sajátságai korlátozzák.)

Az előzőkben, amikor a ki- és beviteli rendszer sajátságait tekintettük át, mindig a kinyomtatás módszereiről beszéltünk. Hasonló elvek szerint képzelendő el az egyéb perifériális egységeken operáló utasítások, és a rájuk vonatkozó formátumutasítások működése is. A beolvasási utasítások esetében a formátumutasítás azt közli, hogy milyen jellegű bemenő adatot várhat a gép, amikor az olvasást végrehajtja. Bemenő adatok esetében nem mindig baj, ha azok nem felelnek meg teljesen a formátumspecifikációnak; a gép elbírálhatja, hogy a beolvasott adat elfogad-

ható-e, vagy nem. Például egy beolvasó utasítás végrehajtható akkor is, ha az (5.44) formátumspecifikáció mellett a beolvasott szám 17.5 — a gép figyelmen kívül hagyhatja a specifikációt, és átalakíthatja a 17.5 számot olyan alakra, ahogyan az a gépben tárolódik. Viszont esetleg figyelmeztetést ad. Ha ellenben pl. számot várt a specifikáció alapján és szöveg jött, akkor már nem tud mihez kezdeni, és „erélyesebben" figyelmeztet a rendellenességre (pl. megáll, vagy automatikusan áttér egy másik feladatra).

Külső memóriákkal való adatcsere esetén szintén van értelme a formátum definiálásának. Az ilyen jellegű adatcserének két alapesete van. Az első esetben a külső memóriában való adattárolás oka, hogy valamely számítás elvégzése közben olyan sok részeredmény képződik, amennyit a gép nem tud a belső memóriában tárolni, és ezeket addig is, amíg további felhasználásukra sor kerül, kiírja valamelyik külső memóriaegységébe, pl. mágnesszalagra, mágnesdobra stb. Ebben az esetben az a célszerű, hogy az átvitel a gép belső számábrázolásának megfelelően történjék, azaz átvitel előtt semmiféle átalakítást (számrendszerváltás, tizedespont elhelyezése stb.) ne végezzen a gép. Nyilván ugyanez érvényes a visszaolvasásra is. Ezt a fajta átvitelt *formátum nélküli átvitelnek* nevezzük.

A külső memóriához való fordulás azonban lehet közvetett kinyomtatás is. Az igen nagy teljesítményű gépek ugyanis a perifériális egységekre való várakozás idejének csökkentése céljából általában nem közvetlenül nyomtatják a számítások végeredményeit, hanem előzőleg átírják egy külső memóriaegységbe, amely közvetlen kapcsolatban áll a nyomtatóberendezésekkel, és az effektív nyomtatás a gép működésétől függetlenül később történik*. Egy ilyen jellegű átvitel kapcsán nyilván mindazt el kell a gépnek végeznie, amit egyébként közvetlen kinyomtatás előtt végezne, tehát ez ugyanolyan formátumvezérlés hatása alatt történik, mint a kinyomtatás. Itt a „külső memória" alatt

* Az ilyen kinyomtatást - szemben a gép működésével egyidőben történő „on line" kinyomtatással „off line" nyomtatásnak nevezi az angol terminológia, és egyelőre a magyar nyelvben is ez a terminus használatos.

érthető adott esetben szalag- vagy kártyalyukasztó berendezés is.

A fentiekből teljesen világosan kiderül, hogy nem kell annyiféle speciális utasítást kitalálni egy ki- és beviteli rendszer megkonstruálásához, ahány perifériális egység van, hanem az egész kérdéskomplexum kezelhető egységesen is. Mindössze annyit kell megjegyeznünk, hogy egyes „végrehajtó" utasítások másképpen reagálnak ugyanarra a formátumspecifikációra, mint mások (pl. ki- és bevitel). Ilyen egységes elveken alapuló rendszer van a FORTRAN nyelvben, és hasonlót javasol a [3] cikk az ALGOL számára is.

A perifériális egységekre hivatkozó utasítások körébe tartoznak azok az utasítások is, amelyeknek a programok kipróbálásában jut fontosabb szerep: a programozó egyes helyeken elhelyezhet nyomtató utasításokat olyan részeredmények kinyomtatására, amelyek csak a program ellenőrzésének stádiumában érdekesek. Az előzőkben kifejtett elvek alapján nyilvánvaló, hogy a formátum meghatározásának része lehet egy olyan kikötés is, mely szerint a nyomtatás csak valamely külső feltétel teljesülése — pl. a gép vezérlőpultján egy kapcsolónak meghatározott állása — esetén hajtandó végre. Itt megjegyezzük, hogy ha a gép lyukkártyás beviteli rendszerű, akkor a programozónak bármikor módja van a számára nem kívánatossá vált utasításokat egyszerűen kiemelni a programból (ez csak néhány kártya kiemelését jelenti), ezért lyukkártyás gépekre az ilyen feltételes nyomtatásoknak nincs különös jelentőségük. Szalagos bevitelű gépeken az utasítások törlése sokkal nehézkesebb, és ott a feltételes nyomtatások lehetősége igen jó segítség a programozónak.

Ezzel áttekintettük a ki- és beviteli rendszerek szerkezetének főbb elveit. Igyekeztünk kiragadni ezekből azt, ami közös, figyelembe véve az irodalmi javaslatokat, valamint a meglevő gépi reprezentánsok ki- és beviteli rendszerének sajátosságait. Természetesen általánosan elfogadott egységes rendszer hiján, egyes konkrét gépi reprezentánsokban a fenti elvek megvalósítása nagymértékben különbözhet. Például több olyan rendszer létezik, amelyben a formátumspecifikáció része a nyomtatási utasításnak

(együtt van a „végrehajtó" és a „vezérlő" rész). Az alapelvek azonban igen hasonlóak, és ezért úgy gondoljuk, hogy azok ismeretében az Olvasónak nem fog nehézséget okozni bármelyik konkrét, vagy egy esetleges később bevezetendő általános rendszer elsajátítása.

## 8. Gyakorlatok

**1.** Írjunk eljárásdeklarációt négyzetes mátrix és vektor szorzatának meghatározására! A formális paraméterek legyenek a következők: a mátrix azonosítója $A$, a rendszám $n$, a vektor azonosítója $v$, az eredményvektor azonosítója $x$. Eljárástörzsként felhasználhatjuk a (3.49) algoritmust.

**2.** Írjunk eljárásdeklarációt mérési ponthalmaznak egycnessel való „kiegyenlítésére" (a ponthalmazhoz egyenes illesztésére)! Az eljárástörzs legyen (3.76); a formális paramétereket válasszuk meg úgy, hogy azok minél jobban megfeleljenek a feladat természetének!

**3.** Muller módszere szerint az $f(z)=0$ transzcendens egyenlet megoldására az alábbi iteratív eljárás alkalmazható: Legyenek adva a gyök $z_0$, $z_1$ és $z_2$ közelitései, valamint az ezen értékekhez tartozó $f_0$, $f_1$ és $f_2$ függvényértékek. E kezdőértékekből kiindulva, és feltételezve, hogy a $z_{i-2}$, $z_{i-1}$ és $z_i$ közelítések már meg vannak határozva, $z_{i+1}$-et a következő iterációval határozhatjuk meg:

$$z_{i+1} = z_i + \lambda_{i+1} h_i, \tag{5.50}$$

ahol

$$\lambda_{i+1} = \frac{-2 f_i \delta_i}{g_i \pm \{g_i^2 - 4 f_i \delta_i \lambda_i [f_{i-2} \lambda_i - f_{i-1} \delta_i + f_i]\}^{1/2}};$$

$$g_i = f_{i-2} \lambda_i^2 - f_{i-1} \delta_i^2 + f_i (\lambda_i + \delta_i);$$

$$h_i = z_i - z_{i-1}; \quad \lambda_i = \frac{h_i}{h_{i-1}}; \quad \delta_i = 1 + \lambda_i.$$

$\lambda_{i+1}$ kifejezésében a $\pm$ előjelek közül azt kell venni, amely mellett $\lambda_{i+1}$ abszolút értéke kisebb.

Az (5.50) iteráció azon alapul, hogy a gyök három utolsó közelitésének megfelelő pontokon át egy másodfokú polinomot fektetünk, és ennek a polinomnak az egyik gyökét tekintjük a megoldás újabb közelitésének. A módszer előnye, hogy a konvergencia általában gyorsabb, mint a közismertebb módszerek esetében. Egy iterációs lépéshez

csak egyszer kell a függvény helyettesítési értékét kiszámítani, és a függvény deriváltja nem kerül felhasználásra, mint pl. a Newton-módszer esetében. Hátránya viszont, hogy csak elegendően jó kezdőértékek mellett konvergens, és az iterációs formulák eléggé bonyolultak. Alkalmazása akkor ajánlható, ha $f$ igen bonyolult függvény, és a gyöknek ismeretes egy jó közelítése (lehetőleg a gyök „el legyen választva", azaz ismeretes legyen egy olyan alsó és felső becslése, amelyek közé az egyenletnek nem esik más gyöke — azonban ez nem szükséges a módszer alkalmazhatóságához). Az iteráció lefolytatható a komplex számsíkon; szokták a módszert komplex algebrai egyenletek megoldására is használni.

Írjunk eljárásdeklarációt Muller módszeréhez, valós aritmetikát alkalmazva. A formális paramétereket válasszuk értelemszerűen. Az iterációt folytassuk addig, amíg az

$$\eta_{i+1} = \frac{h_{i+1}}{z_{i+1}}$$

hányados kisebbé nem válik, mint valamely *epsz* formális paraméter! Ezután írjunk teljes programot, amely megoldja a

$$\sin \frac{1}{z} = 0$$

egyenletet. Ennek az egyenletnek, mint látható, gyöke $\pi$ minden egész számú többszörösének reciproka. Határoztassuk meg a programmal a három legnagyobb pozitív gyököt. Használjuk a következő kezdő közelítéseket:

$$\text{Első gyök} \begin{cases} z_0 = 0.25 \\ z_1 = 0.35 \\ z_2 = 0.45 \end{cases}$$

$$\text{Második gyök} \begin{cases} z_0 = 0.13 \\ z_1 = 0.17 \\ z_2 = 0.205 \end{cases}$$

$$\text{Harmadik gyök} \begin{cases} z_0 = 0.0905 \\ z_1 = 0.1 \\ z_2 = 0.12 \end{cases}$$

Az *epsz* paraméter aktuális értékét válasszuk $10^{-6}$-nak.

**4.** Mint ismeretes, két egész szám legnagyobb közös osztóját az ún. euklideszi algoritmus alapján lehet meghatározni. Ez a következő: legyenek $b_0$ és $b_1$ azok a számok, amelyeknek *LNKO*-ját keressük. Fel-

tesszük, hogy $b_0 > 0$ és $b_1 > 0$. Elosztjuk $b_0$-t $b_1$-gyel, és kapunk valamely $b_2$ maradékot:

$$b_0 = h_1 b_1 + b_2, \qquad \text{ahol} \qquad b_2 < b_1.$$

Megismételjük most ugyanezt $b_1$-gyel és $b_2$-vel:

$$b_1 = h_2 b_2 + b_3, \qquad \text{ahol} \qquad b_3 < b_2$$
$$\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots$$
$$b_{i-1} = h_i b_i + b_{i+1}, \qquad \text{ahol} \qquad b_{i+1} < b_i.$$

Tekintve, hogy $b_1 > b_2 > \dots$, és a $b$ számok pozitív egészek, a fenti sorozat véges sok lépésben 0-ra csökken. Ha $b_{n+1} = 0$, de $b_n \neq 0$, akkor $b_0$ és $b_1$ *LNKO*-ja $b_n$.

Írjunk függvényeljárást az euklideszi algoritmushoz!

**5.** Valamely egyváltozós függvény lokális maximumát a következő algoritmus alapján kereshetjük meg: feltesszük, hogy adott a lokális maximum abszcisszájának valamely $x_0$ kezdeti közelítése, valamint egy $\delta x$ kezdeti lépéshossz. Kiindulva az $x_0$ kezdeti közelítésből, haladjunk az $x$ tengely mentén $\delta x$ lépésközzel mindaddig, amíg a függvényértékek növekszenek. Ha azt észleljük, hogy a függvényértékek csökkenni kezdenek, visszafordulunk, és megismételjük az előzőket, de harmad lépésközzel. Ezt az eljárást addig ismételjük, amíg $\delta x$ elegendően kicsinnyé nem válik. Írjuk le ezt az algoritmust ALGOL eljárásdeklaráció formájában!

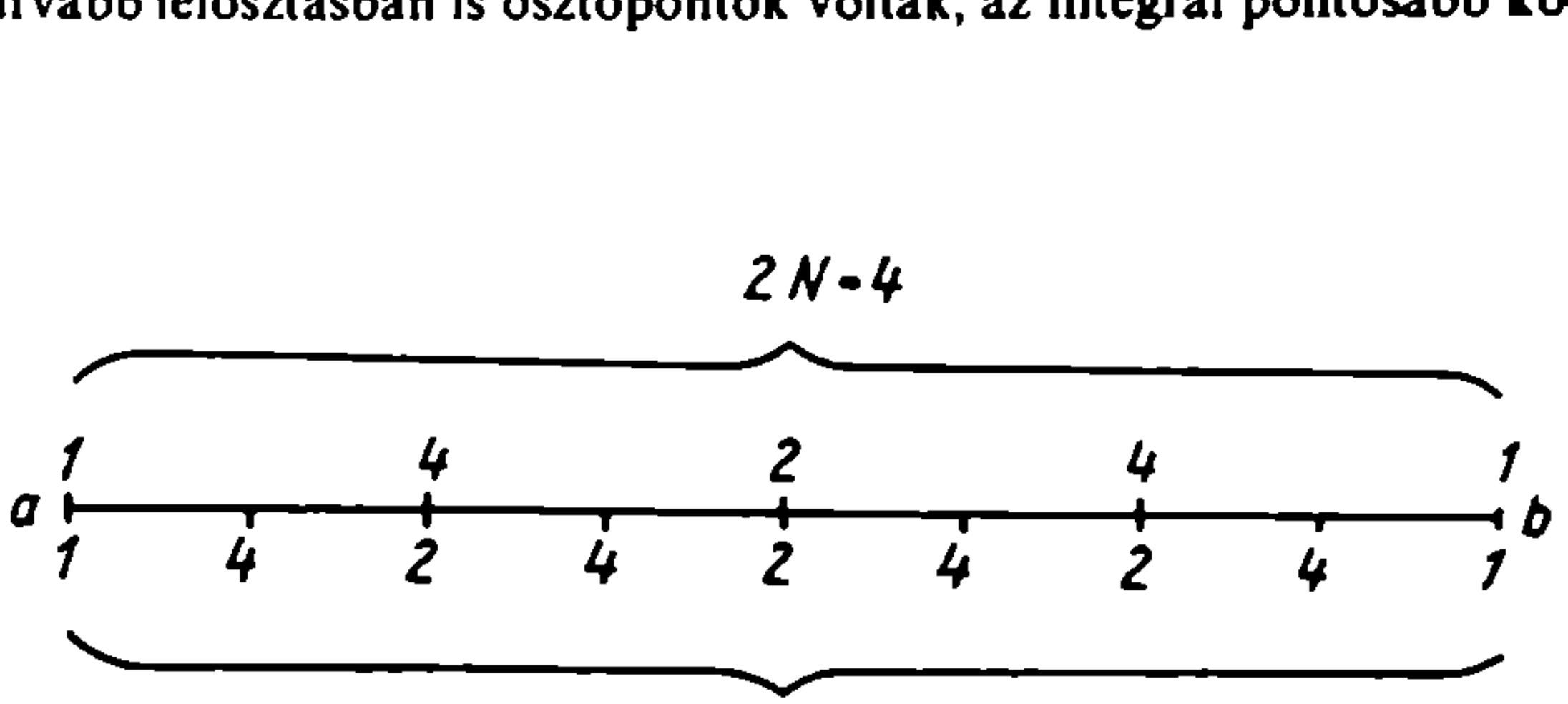
Az algoritmus általánosítható kétváltozós függvény szélsőértékének megkeresésére is. Ebben az esetben úgy járhatunk el, hogy először megkeressük a függvény egy feltételes szélsőértékét az előbbi algoritmus alapján, pl. rögzített $y$ mellett, $x$ szerint. Ezután rögzítjük $x$-et az előbb meghatározott szélsőérték helyén, és ezen rögzített $x$ mellett meghatározzuk a szélsőértéket $y$ szerint. Ezt követőleg megint $y$-t rögzítjük a szélsőérték helyén, és $x$ szerint keressük tovább, s így tovább. Az eljárás addig folytatható, amíg mind $x$-ben, mind $y$-ban elegendően kicsinnyé nem válik a szélsőérték két egymásutáni közelítésének megváltozása.

Mind az egy-, mind a kétdimenziós esetben mérhetjük a megváltozást nemcsak a független változón, hanem a függő változón is, tehát szabhatunk olyan feltételt is az iteráció befejezésére, hogy a függvényértéknek kell elegendően kevéssé változnia. A feladat természete dönti el, melyik a célszerű; ha a további számítás során a szélsőérték helyét használjuk fel, akkor azt kell minél pontosabban meghatároznunk, ha pedig magát a szélsőérték helyén vett függvényértéket, akkor utóbbit.

Írjunk eljárásdeklarációt, amely az egydimenziós esetre alkal-

mazható, és egy másikat, amelyet kétdimenziós esetben alkalmazhatunk. Az egydimenziós deklaráció része lehet a kétdimenziósnak, mint pl. az *RK1L* deklaráció része *RK*-nak. [Lásd az (5.34) eljárásdeklarációt.] Az eljárásból való kilépés feltétele legyen az, hogy az abszcisszákban az eltérés két egymás utáni iterációs lépésben ne legyen nagyobb, mint *epsz*, az ordinátában pedig ne legyen nagyobb, mint *eta*.

6. Írjunk eljárásdeklarációt tetszőleges függvények Simpson-formula szerinti integrálására, automatikus lépésköz-megválasztással! Magát a Simpson integrációs formulát lásd (4.18) alatt; automatikus lépésköz-megválasztás nélküli változatát kidolgoztuk a 4. p-ban. Programozási szempontból itt is, mint (5.34) esetében, a szukcesszív felezés látszik célszerűnek. A felezéses eljárás a következők miatt mutatkozik igen jól programozhatónak: tegyük fel, hogy meghatároztuk az integrál közelítő értékét oly módon, hogy az integrációs intervallumot $2N$ részre osztottuk fel (a felosztásnak mindig páros számú részintervallumra kell történnie), és ki akarjuk számítani az integrál értékének egy pontosabb közelítését $4N$ részre vett felosztás mellett. Két dolgot veszünk észre: 1. a durvább felosztás minden egyes osztópontja osztópont a finomabb felosztás esetén is; 2. azokban a pontokban, amelyek a durvább felosztásban is osztópontok voltak, az integrál pontosabb közelítésének számításakor 2-vel kell a függvényértéket szorozni, míg azokban a pontokban, amelyek a durvább felosztás mellett nem voltak osztópontok, 4-gyel. [A lépésköz egyharmadát az összeg elé kiemelve képzeljük, lásd a (4.19) formulát.] A 6. ábrán a durvább és a finomabb felosztást és a megfelelő együtthatókat tüntettük fel. Látjuk, hogy az intervallum kezdő- és végpontja kivétel, mert az minden felosztás mellett osztópont, és a függvényértéket ott minden esetben 1-gyel kell szorozni.




6. ábra.

A fenti észrevételek alapján nyilvánvaló, hogy az újabb közelítés számításához nem kell minden pontban kiszámítanunk a függvényértéket, hanem csak a „közbülső" pontokban, tehát azokban, amelyekben a

függvényértéket 4-es szorzóval kell venni. Jelentse $S_1^{(4N)}$ azon pontokban vett függvényértékek összegét, amelyeket a finomabb felosztásban 2-es szorzóval kell venni, $S_2^{(4N)}$ azon pontokban vett függvényértékek összegét, amelyeket 4-es szorzóval kell venni, $S_3^{(2N)}$ a durvább felosztás valamennyi osztópontjában vett függvényértékek összegét, a kezdő és a végpontban vett függvényértékek kivételével, végül $S_3^{(4N)}$ ugyanezt a finomabb felosztás esetén. Fennállnak a következő összefüggések:

$$S_3^{(2N)} = S_1^{(4N)}, \quad S_3^{(4N)} = S_1^{(4N)} + S_2^{(4N)}, \tag{5.51}$$

$$I^{(4N)} = \frac{b-a}{12N}\{2S_1^{(4N)} + 4S_2^{(4N)} + f(a) + f(b)\}.$$

Az (5.51) formulák segítségével iterativ eljárást definiálhatunk; finomabb felosztás esetén mindig csak az $S_2$ szummát kell újonnan meghatároznunk, az $S_1$ szumma az előző felosztás alapján adva van. Mindkettőt kiszámítani csak a kezdeti felosztásnál kell. Az iteráció abbahagyható, ha valamely $n$-re teljesül az

$$\left|\frac{I^{(4N)} - I^{(2N)}}{I^{(4N)}}\right| < \varepsilon$$

egyenlőtlenség.

Megjegyczzük még, hogy ha ismeretes valamely $N$-re $I^{(4N)}$ és $I^{(2N)}$, akkor a numerikus analízis tankönyvei szerint az

$$\tilde{I}^{(4N)} = I^{(4N)} + \frac{16}{15}\{I^{(4N)} - I^{(2N)}\} \tag{5.52}$$

mennyiség bizonyos feltételek mellett jobb közelítését adja az integrál valódi értékének, mint $I^{(4N)}$. A programba esetleg ezt is beleépíthetjük.

7. Példák rekurzív eljárásokra:

*a)* Polinom helyettesítési értékének meghatározása a Horner-séma alapján. Mint ismeretes, a Horner-sémát rekurzív alakban a következőképpen lehet felírni: legyen $P_n(x)$ maga az $n$-edfokú polinom, és $P_{n-1}(x)$ az az eggyel alacsonyabb fokú polinom, amelyet úgy kapunk, hogy elhagyjuk $P_n(x)$ konstans tagját, és minden egyes hatványkitevőt eggyel csökkentünk. $P_n(x)$-re a következő rekurzív formula érvényes:

$$P_n(x) = P_{n-1}(x) \times x + a_n, \tag{5.53}$$

ahol $a_n$ a $P_n(x)$ polinom konstans tagja. (5.53) segítségével egy $n$-edfokú polinom kiszámítását visszavezettük egy $(n-1)$-edfokú polinom kiszámítására. Írjuk le ezt az eljárást rekurzív ALGOL-procedura formájában!

*b)* Írjunk ALGOL-programot a következő számsorozat *n*-edik elemének kiszámítására:

$$a_{n+1} = \frac{1}{a_n} + \frac{1}{a_{n-1}},$$

ahol $a_0 = 2$, $a_1 = 5$.

*c)* Írjunk rekurzív eljárást *n*-szeres integrálok kiszámítására. Egy *n*-dimenziós téglatesten vett integrál kiszámítására az alábbi rekurzív összefüggés használható:

$$\int_{T_n} f(x)dT_n = \int_{a_n}^{b_n} \left\{ \int_{T_n-1} f(x)dT_{n-1} \right\} dx_n. \tag{5.54}$$

Az algoritmus egyszerűsítése érdekében helyettesítsük az integrált közönséges téglányösszeggel. Az (5.54) formulában $T_n$ az $\{a_1b_1, a_2b_2, \ldots, a_nb_n\}$ *n*-dimenziós téglatestet jelenti, $T_{n-1}$ pedig ennek az első $n-1$ koordinátatengely által kifeszített altérre vetett vetületét, vagyis az $\{a_1b_1, a_2b_2, \ldots, a_{n-1}b_{n-1}\}$ $(n-1)$-dimenziós téglatestet.

**8.** Tekintsük az alábbi ALGOL-blokkot:

```
begin comment a blokknak két globális azonosítója van, N és P. P egy-
méretű tömb, N jelenti P elemeinek a számát;
   procedure GO (OPER, COLL); procedure OPER, COLL;
      Eljárástörzs: COLL (OPER);
   procedure COLLECT (OPERATÍV); procedure OPERATIV;
      begin integer i;
      for i := 1 step 1 until N do OPERATIV (P[i]) end COLLECT;
procedure DO (S); real S; S := 0;                                  (5.55)
comment A blokk törzse egyetlen utasításból áll;
   GO (DO, COLLECT) end
```

Állapítsuk meg az (5.55) programról, hogy mit hajtat végre! Ha már értjük működését, akkor a lehető legkevesebb átalakítással írjuk át úgy, hogy az átalakított program

*a)* Keresse meg *P* maximális abszolút értékű elemét;

*b)* Írjon zérust a blokkra nézve ugyancsak globálisnak feltételezett *A*, *B* és *C* változókba. (Félreértések elkerülése végett: *a)* és *b)* teljesen független feladatok!)

Útmutatás: a program egy kis „bűvészkedés" a név szerinti paraméterhívással. Alkalmazzuk gondosan a 2. p-ban mondottakat az eljárástörzs „kitöltéséről", és hamarosan rá fogunk jönni a megoldásra.

# FÜGGELÉK

## A metanyelvi leírás módszere

A programozáselméleti szakirodalomban a programozási nyelvek szintaktikus szabályait általában az ún. *metanyelvi módszer* segítségével szokták leírni, amelyet *J. W. Backus* dolgozott ki. Ezt a módszert követik az ALGOL-jelentések is, de hasonló formában adja meg pl. a [18] cikk is a szabványos ki- és beviteli eljárások javasolt szintaktikus szabályait. Ebben a függelékben röviden ismertetjük a metanyelvi leírási módszert azok számára, akik ismereteik további bővítése céljából programozáselméleti szakirodalmat kívánnak tanulmányozni.

A metanyelv bevezetését az a körülmény tette szükségessé, hogy az emberi beszéd nyelvei, az ún. természetes nyelvek nem eléggé szabatosak egy programozási nyelv formális szabályainak megfogalmazásához, illetőleg a szabályok szabatos leírása igen terjedelmes szövegezést igényel. A nagyobb tömörség és szabatosság érdekében a definiálandó programozási nyelvet, pl. az ALGOL-t egy „közbülső nyelv", a metanyelv jelöléseivel írják le. Az elnevezés arra utal, hogy a természetes nyelv és a definiálandó „célnyelv" közé iktatott harmadik nyelvről van szó, amelynek segítségével a célnyelvet definiáljuk.

A metanyelvi leírásban az egyes szintaktikus egységek definícióit ún. *metanyelvi egyenlőségek* segítségével adjuk meg. A metanyelvi egyenlőség bal oldalán a definiálandó szintaktikus egység elnevezése áll, metazárójelbe zárva (a metazárójelek a következő alakúak: ⟨,⟩); ezzel azt fejezzük ki, hogy valamennyi olyan jelsorozatra gondolunk, amely az adott szintaktikus egységbe tartozik. A metanyelvi egyenlőségekben az egyenlőségjel ::= (olvasva: négypont-

egyenlő). A metanyelvi egyenlőség jobb oldalán a definiáló szintaktikus egységek állnak alkalmas módon kódolva.

Az alapjelek a metanyelv szempontjából egyelemű szintaktikus egységek, melyek egyetlen eleme maga az alapjel. Pl. a < + > szintaktikus egységet, melynek egyetlen eleme a + alapjel, lehetne metazárójelbe is zárni, azonban a metazárójelek számának csökkentése érdekében ettől eltekintenek, és az alapjelek a metanyelvi egyenlőségekben önmagukban, metazárójel nélkül állnak.

Az egyik legfontosabb fogalom, amellyel a metanyelvi egyenlőségek dolgoznak, egy adott jelsorozat *folytatása*. Ha két szintaktikus egység nevét, mindkettőt metazárójelbe zárva, egymás mellé írjuk, ezen azt értjük, hogy tekintünk minden olyan alapjel-sorozatot, amely úgy jön létre, hogy egy tetszőleges olyan jelsorozatot, amely az első szintaktikus egységbe tartozik, folytatunk egy tetszőleges olyan jelsorozattal, amely a második szintaktikus egységbe tartozik. Ezt a műveletet *juxtapozíciónak* (melléhelyezés) nevezzük, és alkalmazhatjuk többször egymás után is. Miután az alapjelek, mint mondottuk, nem kerülnek metazárójelbe, ha azt akarjuk kifejezni, hogy egy jelsorozatot valamilyen meghatározott alapjellel folytatunk, akkor magát az alapjelet írjuk a metazárójelbe írt kategórianév után.

Mielőtt a fentiekre példát mutatnánk, be kell vezetnünk még egy metanyelvi fogalmat. A függőleges vonalka a metanyelvben az „akár" szót helyettesíti. A metanyelvi egyenlőségek jobb oldalán nem is használnak más jelöléseket, mint a juxtapozíciót és függőleges vonalkát.

Lássunk most néhány egyszerű példát metanyelvi egyenlőségre! Első példaként tekinthetjük a <számjegy> szintaktikus egység definícióját:

$$<\text{számjegy}> ::= 0|1|2|3|4|5|6|7|8|9 \qquad \text{(F.1)}$$

Az (F.1) definíció szerint, a <számjegy> szintaktikus egységbe a 0, 1, 2, ..., 9 alapjelek tartoznak. Más szóval, (F.1) azt mondja ki, hogy a továbbiakban bárhol, ahol számjegyről beszélünk, ezen a 0, 1, 2, ..., 9 jelek bármelyikét értjük.

Kissé bonyolultabb példa a következő:

<azonosító> ::= <betű> | <azonosító> <betű>
<azonosító> <számjegy> (F.2)

Az (F.2) definíció a következőképpen oldható fel szavakban:

1. Azonosító lehet bármely magában álló betű. (Előzetesen már definiálva van, hogy mely alapjelek tartoznak a betű szintaktikus egységbe.)

2. Azonosító lehet ezenkívül bármely jelsorozat, amelyet úgy kapunk, hogy egy az <azonosító> szintaktikus egységbe tartozó jelsorozatot betűvel folytatunk. Más szóval, egy azonosítóhoz egy betűt hozzáírva, újabb azonosítót kapunk, betű hozzáírása nem vezet ki a szintaktikus egységből.

3. Hasonlóképpen számjegy hozzáírása sem vezet ki a szintaktikus egységből. [A számjegyek is előzetesen már definiálva voltak az (F.1) metanyelvi egyenlőség segítségével.]

Az (F.2) metanyelvi egyenlőség vizsgálata közben egy fogalmilag új dologhoz érkeztünk el. (F.2) jobb oldalán ugyanis a bal oldali, definiálandó szintaktikus egység neve fordul elő. Ez nem logikai cirkulus vitiosus, önmagával való definíció, hanem rekurzív műveleti utasítás arra nézve, hogy ha már tudjuk egy jelsorozatról, hogy az <azonosító> szintaktikus egységbe tartozik, akkor hogyan lehet ebből az egyetlen jelsorozatból tetszőleges számú további azonosítót előállítani. Igen fontos szerepe van az (F.2) jobb oldalán álló <betű> szónak, mert ez ad felvilágosítást arra nézve, hogyan lehet találni olyan jelet, amelyről biztosan tudjuk, hogy azonosító. Enélkül (F.2) értelmetlen volna, mert csak azt mondaná ki, hogy hogyan lehet egy azonosítót a szintaktikus egységből való kilépés nélkül folytatni, de semmi sem biztosítaná, hogy van akár egyetlen jelsorozat is, amely egy ilyen folytatási eljáráshoz kiindulásul szolgálhatna.

A fentiek alapján könnyű belátni, hogy az (F.2) definíció azt az ismert formális szabályt adja, hogy az ALGOL-

ban azonosító lehet minden, betűvel kezdődő és betűkből és számjegyekből álló jelsorozat. Ennek belátásához csak az 1—3. alattiakat kell ismételten alkalmazni.

Az (F.2) definíciót *rekurzív definíciónak* nevezzük. A rekurzív definíció alapgondolata a következő: kiindulva egy jelsorozatból, amelyről valahonnan tudjuk, hogy a definiálandó szintaktikus egységbe tartozik, különböző jelek hozzávételével ismét olyan jelsorozatot állítunk elő, amely ugyanebbe a szintaktikus egységbe tartozik. Ahhoz természetesen, hogy a definíció értelmes legyen, biztosítani kell olyan jelsorozat létezését, amelyből ki lehet indulni, amint azt az (F.2) definíció esetében is láttuk.

Az ALGOL-t leíró metanyelvi egyenlőségrendszer főként rekurzív definíciókat tartalmaz. A definíció rekurzív jellege azonban nem mindig nyilvánul meg olyan világosan, mint az (F.2) definíció esetében láttuk. Az alapgondolat ugyanaz: valami módon definiálásra kerül a meghatározandó szintaktikus egység egy alapesete, és ebből kiindulva kell, adott eljárás szerint, felépíteni az általánost. Azonban a kiindulásul vett alaptípus definiálására általában nem elég egyetlen metanyelvi egyenlőség, hanem azt magát is több egyenlőség segítségével definiálják. Ilyen módon egy több egyenlőségből álló metanyelvi egyenlőségrendszer adódik, azzal a sajátsággal, hogy egyes egyenlőségekben olyan szintaktikus egységek neve szerepel, amelyeket ugyanez az egyenlőségrendszer csak néhány sorral alább definiál. Az ilyen rendszerek feloldásában ugyanazt az elvet kell követni, amelyet az (F.2) definíciónál láttunk.

A teljes ALGOL 60 szintaxis ilyen módon való megértése azonban igen nehézkes, ezért e könyvben inkább a terjedelmesebb, de érthetőbb utat választottuk, a szintaktikus definícióknak szavakban való feloldását.

# A GYAKORLATOK MEGOLDÁSAI

## Második fejezet

**1.** *a)* $(a \times sin(2 \times n \times x) + b \times cos(2 \times n \times x))/(a \times x\uparrow 2 + 2 \times b \times x \times y + c \times y\uparrow 2)$

*b)* $((a+b)\uparrow 3 + ln(x\uparrow 2 + y\uparrow 2))/(x-y)\uparrow 0.3333333/exp(-(x\uparrow 2 - y\uparrow 2)/2/alfa\uparrow 2)$

*c)* $1/(1+1/(1+1/(1+1/(1+x))))$

*d)* $(alfa+beta)\uparrow(1/n) \times (gamma-delta)\uparrow(1/m)/(alfa+gamma)\uparrow(1/l)/(beta-delta)\uparrow(1/k)$

**2.** *a)*
$$\frac{q^{prs}x^{uvz}}{a^{bcd}};$$

*b)*
$$\frac{\sqrt{x+y+z}\left(\sqrt[4]{a^4+b^4}+\sqrt[4]{c^4+d^4}\right)}{\sqrt{1+e^{-y(a+2b)+z^2b}}};$$

*c)*
$$\arctan\left(\frac{1+\ln\left(\frac{x^2+y^2-z^2}{p^2-q^2+s^2}\right)}{1+e^{\frac{b^2+c^2-a^2}{d^2+e^2-f^2}}}\right).$$

**3.** *a)* Valamennyi zárójel fölösleges.

*b)* $x\uparrow 2 \times y\uparrow 2/sqrt(x-2-a\uparrow b\uparrow c)$

*c)* $a/(b-c+d \times sin((2 \times a+3 \times b)/(b-2 \times c)))$

**4.** *a)* 2*a* és 2*b*. Ezek sem mint azonosítók, sem mint szorzatok nem értelmezhetők. Azonosító ugyanis nem kezdődhet számjeggyel, szorzatból pedig nem hagyható el a szorzásjel.

*b)* *sin* x, *cos* x. Standard függvény argumentuma nincs zárójelben.

*c)* $_2 - 4$. Nem definiált szimbólum szerepel a kifejezésben.

*d)* ±

*e)* Szögletes zárójel meg nem engedett módon való alkalmazása.

*f)* 2. Szintaktikusan hibás számkonstans.

*g)* $3_{10}i$. Szintaktikusan hibás számkonstans.

A decimális normálótényező ($_{10}$) jele után csak egész számkonstans állhat, változó nem. Ilyen esetben a kifejezést a hatványozás műveleti jelének alkalmazásával kell felírni: *alfa* – *beta*/*gamma* + 3 × 10↑*i* + *j*

5. *a)* 1032.85

*b)* 2.8141593

*c)* Nem értékelhető. (÷ bal oldalán valós típusú mennyiség áll.)

*d)* Nem értékelhető. (*sqrt* művelet negatív argumentumra fordul elő, hatványozás pedig negatív alap és valós kitevő mellett.)

*e)* Nem értékelhető. (A *sqrt* standard függvény mindig valós típusú eredményt állít elő, tehát a ÷ művelet bal oldalán valós típusú mennyiség áll.) Értékelhető kifejezést kapunk, ha a négyzetgyökvonás eredményére az *entier* standard függvényt alkalmazzuk:

$$entier\ (sqrt\ (i+j)) \div k$$

6. Tekintsük a következő aritmetikai kifejezéseket:

$$i - i \div 2 \times 2 \qquad \text{(G.1)}$$

$$1 - i + i \div 2 \times 2 \qquad \text{(G.2)}$$

A (G.1) kifejezés értéke 0, ha *i* páros, és 1, ha *i* páratlan. A (G.2) kifejezés éppen fordítva, 1, ha *i* páros, és 0, ha *i* páratlan. (G.1) és (G.2) segítségével mindhárom feladat könnyen megoldható:

*a)* $f_1(i) = 1 + i \div 2 \times 2$

*b)* $f_2(i) = 2 \times (i - i \div 2) - 1$

*c)* $f_3(i) = 2 \times (i - i \div 2)$

7. A

$$k - k \div n \times n \qquad \text{(G.3)}$$

aritmetikai kifejezés a $k$ számnak $n$-re vonatkozó maradékát állítja elő. Például $k = 14$, $n = 6$ esetén $14 - 14 \div 6 \times 6 = 2$.

Az *a)* gyakorlatban szereplő $f_1(i)$ kifejezés nem egyéb, mint $i$-nek $m$-re vonatkozó maradéka, plusz 1, vagyis

$$f_1(i) = i + 1 - i \div m \times m$$

A *b)* gyakorlatban szereplő $f_2(i)$ kifejezés $(i + m - 2)$-nek $m$-re vonatkozó maradéka, plusz 1. Így

$$f_2(i) = i + m - 1 - (i + m - 2) \div m \times m$$

8. Az *a)* gyakorlat követelményeinek megfelelő szám $(i + j - 1)$-nek $m_1$-re vonatkozó maradéka, plusz 1. A *b)* gyakorlat pedig úgy állít-

ható elő az *a)* gyakorlatból, hogy *j* helyébe $(m-j)$-t írunk. Így

$$f_1(i) = i+j-(i+j-1) \div m \times m$$
$$f_2(i) = i+m-j-(i+m-j-1) \div m \times m$$

Az eredményt verifikálhatjuk az előző feladat megoldásán, ha *j* helyébe 1-et írunk.

**Megjegyzés:** A 7. és 8. gyakorlatok ún. *ciklikus permutálást* végeznek. Ha az *i* változó végigfutja az 1, 2, ..., *m* értékeket, akkor a 7. gyakorlat $f_1(i)$-je a 2, 3, 4, ..., *m*, 1 értékeket futja végig, a 8. gyakorlat $f_1(i)$-je a $j, j+1, j+2, \ldots, m, 1, 2, \ldots, j-1$ értékeket, a 7. gyakorlat $f_2(i)$-je az $m, 1, 2, \ldots m-1$ értékeket, és végül a 8. gyakorlat $f_2(i)$-je az $m-j+1$, $m-j+2$, $m-j+3$, ..., *m*, 1, 2, ..., $m-j$ értékeket.

9. Nyilván olyan aritmetikai kifejezés felírására van szükség, amelynek értéke megmutatja, hogy hányszor lehet *a*-tól *x*-ig a számegyenesre felmérni egy részintervallum $(b-a)/n$ hosszát. Ez a kifejezés a következő:

$$entier\ (n \times (x-a)/(b-a))$$

Annak az intervallumnak az alsó határa, amelyikben *x* benne van:

$$a + entier\ (n \times (x-a)/(b-a)) \times (b-a)/n$$

10. *a)* **true**
*b)* **true**
*c)* **false**
*d)* **true**
*e)* **true**

11. *a)* $(A \equiv B) \vee x \lceil 2 \times p \times q - 3.5_{10} 2 > 0 \supset \neg A \wedge \neg B \vee \neg C \wedge \neg D \equiv C \wedge D$
*b)* $(A \vee B \supset C \wedge \neg A \vee \neg B \supset C) \wedge D$
*c)* Valamennyi zárójel fölösleges.
*d)* Valamennyi zárójel szükséges.

12. $j = j \div i \times i$

## Harmadik fejezet

1. *a)* Komplex szorzás programja:

```
x3:=x1 × x2 − y1 × y2;
y3:=x1 × y2 + y1 × x2; {folytatás}
```

*b)* Komplex osztás programja:

```
 m:=x2↑2 + y2↑2;
x3:=(x1 × x2 + y1 × y2)/m;
y3:=(y1 × x2 − x1 × y2)/m; {folytatás}
```

**2. Komplex számból való négyzetgyökvonás az alábbi formulák segítségével végezhető:**

$$u = \sqrt{\frac{x+\sqrt{x^2+y^2}}{2}}, \qquad v = \pm\sqrt{\frac{-x+\sqrt{x^2+y^2}}{2}}.$$

A gyök alatti gyök előjele mindig pozitív; a külső gyök előjelét $y$ előjelétől függően kell választani. Ha $y \geqq 0$, akkor mind $u$, mind $v$ pozitív előjelű. Ha $y < 0$, akkor $u$ pozitív, $v$ negatív. Mint ismeretes, a négyzetgyökvonás kétértékű művelet, a második gyök az elsőből oly módon áll elő, hogy $u$ és $v$ előjelét egyidejűleg megfordítjuk. Programunk csak az egyik gyököt állítja elő, ha a program későbbi részében mindkét gyökre szükség volna, úgy a megfelelő előjelfordítást az adott helyen kell elvégezni. A program:

```
m:=sqrt(x↑2+y↑2);   u:=sqrt((x+m)/2);
v:=sqrt((-x+m)/2);  v:=if y≧0 then v else -v;
{folytatás}
```

**3. Annak érdekében, hogy a programban kevesebb logikai feltétel szerepeljen, két aritmetikai kifejezést konstruálunk, amelyek zárt formulával definiálnak egy-egy szakaszonként változó függvényt:**

$$abs(sign(y)) - sign(y) = \begin{cases} 2, & \text{ha} \quad y<0 \\ 0, & \text{ha} \quad y\geqq 0 \end{cases}$$

**és**

$$1 - sign(y)/2 = \begin{cases} \frac{3}{2}, & \text{ha} \quad y<0 \\ 1, & \text{ha} \quad y=0 \\ \frac{1}{2}, & \text{ha} \quad y>0 \end{cases}$$

Ezeknek felhasználásával az alábbi program írható fel:

```
pi:=3.141593;
if x=0 then fi:=if y=0 then 0 else pi×(1-sign(y)/2)
   else begin m:=arctan(y/x);
   fi:=if x<0 then m+pi else (abs(sign(y))-sign(y))×pi
   +m end;
ro:=sqrt(x↑2+y↑2);   {folytatás}
```

**4.** A gyakorlat példa az indexen belüli index alkalmazására. Ilyen példát a szövegben nem közöltünk. A feladat megoldásának programja a következő lehet:

```
k:=1;
for j:=1, j+1 while I[j]≠0∨J[j]≠0 do
   begin m:=R[I[j]]:=R[I[j]]-K[j];   R[J[j]]:=R[J[j]]+
   B[j];
      if m<Z[I[j]] then begin U[k]:=m;   V[k]:=I[j];
      k:=k+1 end end;
```

{folytatás}

Azt a körülményt, hogy „nincs több adat", az *I*, ill. *J* vektorok végére elhelyezett zérusokkal jelezzük. A program akkor ért véget, ha mind az *I*, mind a *J* vektor soron következő eleme zérus.

**5.** A gyakorlat megoldása lehet a következő program:

```
for j:=1 step 1 until n do begin nn[j]:=0;   S[j]:=0 end;
for k:=1 step 1 until N do begin
   for i:=1, i+1 while v[k]>t[i] do j:=i;
   S[j]:=v[k]+S[j];   nn[j]:=nn[j]+1 end;
for j:1 step 1 until n do if nn[j]≠0 then S[j]:=S[j]/nn[j];
```

{folytatás}

Az *n* elemű *nn* vektorban a program végrehajtása után megmaradnak az egyes kategóriákba tartozó *v*-beli elemek számai. A program utolsó része — a számtani közepek képzése céljából — elosztja az egyes kategóriákba tartozó elemek összegét azok számával. Mivel elvileg elképzelhető, hogy egyes kategóriákba egyetlen vektorelem sem esik, be kellett iktatnunk egy „védelmet" a programba, mert különben előfordulhatott volna, hogy végrehajtás közben a gép 0-val való osztás következtében megáll.

**6.** *a)* A (3.37) program:

```
S:=0;   i:=0;
for i:=i+1 while i≦n do S:=S+a[i]×b[i];   {folytatás}
```

*b)* A (3.38) program:

```
j:=0;
for j:=j+1 while j≦n do begin
   S:=0;   i:=0;
   for i:=i+1 while i≦n do S:=S+A[j, i]×b[i];   c[j]:=S
   end;   {folytatás}
```

Itt megjegyezzük, hogy a kezdőérték beállítása beleolvasztható a **for** ciklusba, mi csak azért nem tettük ezt, hogy a kapott programváltó-

zatok szemantikailag tökéletesen egyezzenek a (3.48) és (3.49) programokkal. Ha a

**for** $i:=1, i+1$ **while**

verziót alkalmazzuk, akkor — bár gyakorlatilag egyenértékű programot kapunk — ennek szemantikája elvileg némileg eltérne az eredeti programokétól.

7. A (3.87) program érdekes gyakorlati példa lenne olyan **for** ciklusra, ahol a ciklus magja üres utasítás. Sajnos a példa hibás, mert ugyan kiszámítja az $A$ szám négyzetgyökét, de a ciklusból való kilépés után azonnal el is veszíti, miután az eredmény a ciklusváltozóban van tárolva.

8. A program:

```
i:=1;  pi:=3.141593;  m:=0;  delta x:=0.005;
for xx:=0 step 0.005 until pi/4-0.005, pi/4 step delta x until
pi/2-0.0001 do
  begin if xx=pi/4 then m:=1;  x[i]:=xx;  f[i]:=m;
  if xx≧pi/4 then begin
    L : m:=sin(xx+delta x)/cos(xx+delta x);
    if m-f[i]>10-2 then begin delta x:=delta x/2;  go to L
    end end
  else m:=sin(xx+0.005)/cos(xx+0.005);  i:=i+1 end
{folytatás}                                              (G.4)
```

Ez jóval nehezebb feladat az előzőknél. A (G.4) program lényegében egy általános típusú ciklusutasítás, melynek ciklusfejében két **step until** elem szerepel. Az elsőnél a **step** alapjel után szám (0.005) áll, a másodiknál viszont változó, amelynek értékét a ciklus magja menet közben módosítja. Magának a lépéshossz-rövidítésnek a megszervezését egy belső ciklus végzi, melyet egyszerűség kedvéért feltételes, és nem **for** utasítással programoztunk. A programban szereplő számkonstansok értékét előre is kiszámíthatjuk (pl. a pi/2−0.0001 konstanst), ilyen formában csak az áttekinthetőség kedvéért írtuk azokat. A program működése során igen nagy számú függvényérték képződik, melyeket gyakorlati program esetén nem tárolnánk, hanem azonnal nyomtatnánk.

9. A megoldásnak két lehetséges változatát mutatjuk be. Az első:

```
for j:=2 step 1 until 6 do c[j]:=1;  P:=1;  Q:=0;
c[1]:=0;
for P:=P while P<6 do begin c[Q+1]:=c[Q+1]+1;
  Q:=i:=0;  B[c[6], c[5], c[4], c[3], c[2], c[1]]:=0;
  for i:=i+1 while c[i]=n do begin c[i]:=1;  Q:=i end;
  if Q>P then P:=Q end
{folytatás}
```

A program alapgondolata a következő. A *c* azonosítóval ellátott hatelemű vektor elemei számlálórekeszként működnek. A *c*[1], *c*[2], *c*[3], *c*[4], *c*[5], *c*[6] számhatos adja meg a soron következő tömbelem indexrendszerét.* A *B* tömb elemeit lexikongrafikus sorrendben írjuk tele zérusokkal. Az elemek számlálása úgy történik, hogy egy ciklus 1-re állítja az utolsó *Q* darab számlálórekeszt, ahol *Q* azon számlálórekesz sorszáma, amelytől kezdve egészen az utolsóig (amely az 1-es sorszámot viseli) mindegyik tartalma *n*-nel egyenlő. Ha $c[1] \neq n$, akkor $Q = 0$. Ha például $c[1] = c[2] = c[3] = n$, de $c[4] \neq n$, akkor az említett ciklus 1-et ír *c*[1]-be, *c*[2]-be és *c*[3]-ba. Ezután a $Q + 1$ indexű számlálórekeszt 1-gyel megnöveljük; ha $Q = 0$, akkor *c*[1]-et, majd zérust írunk a megfelelő tömbelem helyére. A *P* nevű változóban jegyezzük azt a maximális *Q* értéket, amely az algoritmus folyamán előfordult. Az algoritmus akkor ér véget, amikor *P* elérte a hatot. *Q* értéke nem nő monoton az algoritmus közben, de *P*-é igen.

Az előző feladatot megközelíthetjük más oldalról is. Az alábbiakban egy olyan megoldást közlünk, amely az elemi számelmélet néhány ismert tételére támaszkodik.

Ismeretes az elemi számelméletből, hogy bármely *K* pozitív egész szám, amely kisebb, mint valamely *P* pozitív egész szám *m*-edik hatványa, egy és csakis egyféleképpen előállítható a következő alakban:

$$K = \sum_{i=0}^{m-1} k_i P^i,$$

ahol a $k_i$ együtthatók 0 és $P - 1$ közé esnek. A fenti szummát a *K* szám *P* alapú számrendszerbeli előállításának nevezzük. A gyakorlati életben a tizes alapú számrendszert használjuk. Az elektronikus számológépek általában a bináris (kettes alapú) számrendszerben dolgoznak. Ismeretes az is, hogy ha *K* végigfut a 0, 1, 2, ..., $P^m - 1$ számokon, akkor a hozzá tartozó *k* együtthatók egymástól függetlenül felveszik a 0 és $P - 1$ közé eső valamennyi értéket.

A *K* szám adott felírásából az is látható, hogy az utolsó $m - 1$ tag összege mindig osztható *P*-vel (*P* kiemelhető), az első együttható, $k_0$ éppen a teljes összeg *P*-vel való osztása esetén fellépő maradék. Ha az első tagot elhagyjuk, és a maradék összegét osztjuk *P*-vel (azaz minden tagban *P* kitevőjét 1-gyel csökkentjük), ismét olyan összegre jutunk, amely az első tagja kivételével osztható *P*-vel, és az első együttható — mely most már $k_1$ — megint csak (a már egyszer redukált)

* Ha a feladatot hatszoros ciklussal oldjuk meg, akkor hat darab ciklusváltozóra van szükség, és a ciklusváltozók aktuális értékei határozzák meg a soron következő tömbelemet. A *c* vektor elemei a hat ciklusváltozót pótolják.

teljes összegnek *P*-vel való osztása esetén fellépő maradéka. A fenti eljárás mindaddig folytatható, amíg az összeg tagjai el nem fogynak.

Ez az algoritmus a következő számítási eljárásra vezet:

1. Ha valamely *K* számot osztunk *P*-vel (*K* és *P* pozitív egészek), az osztás esetén fellépő maradék éppen a *K* szám *P* alapú előállításának utolsó számjegye, $k_0$.

2. Ha a hányados egész részét ismét osztjuk *P*-vel, a maradék adja az előállítás soron következő számjegyét, $k_1$-et.

3. Az új hányados egész részének és *P*-nek hányadosa adja a következő számjegyet, $k_2$-t. Az eljárás akkor ér véget, amikor a hányados egész része zérussá válik.

Ha mármost a *K* szám végigfutja az 1, 2, ..., $P^m - 1$ számokat, akkor a fenti algoritmus segítségével megkapjuk a $k_i$ együtthatók összes *m* elemű (ún. *m*-edosztályú ismétléses) variációját.

Eredeti gyakorlatunkra visszatérve, ott éppen ilyen jellegű feladattal állunk szemben: fel kell sorolnunk a *B* tömb összes lehetséges indexhatosát, vagyis a *c*[1], *c*[2], ..., *c*[6] „számlálóknak" egymástól függetlenül fel kell venniük az 1, 2, ..., *n* értékeket. Az indexhatosok előállítására tehát alkalmazható a fenti számelméleti meggondolások alapján nyert algoritmus: valamely *K* számot végig kell futtatnunk a 0, 1, ..., $n^6 - 1$ számokon, és ennek *n* alapú előállításában a számjegyek „majdnem" előállítják a kívánt indexhatosokat, csak éppen egyet hozzájuk kell adni. Példánkban tehát $P = n$ és $m = 6$. Elgondolásunkat a következő ALGOL-programmal realizálhatjuk:

```
for K:=0 step 1 until n↑6-1 do
  begin M:=K;   for j:=1 step 1 until 6 do begin
    Q:=M÷n;   c[j]:=M-Q×n+1;   M:=Q end;
  B[c[6], c[5], c[4], c[3], c[2], c[1]]:=0 end {folytatás}
```

A programban a *Q* változó tartalmazza a „mindenkori" hányados egész részét; ezt *n*-nel szorozva és a számlálóból kivonva kapjuk a maradékot, amelyet mindjárt a megfelelő *c*[*j*] indexváltozóba helyezünk. Az algoritmus pontosan követi az előzőkben leírtakat.

Látjuk, hogy — némi számelméleti megfontolások alapján — lényegesen egyszerűbb programra jutottunk, mint az előző variánsnál.

Az első algoritmus könnyen átvihető egy általánosabb feladatra, amikor ti. a hatméretű tömb indexei részére különböző indexhatárok vannak előírva (az első index 1-től $n_1$-ig, a második 1-től $n_2$-ig változhat stb.).* A második algoritmus ilyen vonatkozásban

* Az Olvasóra bízzuk az első program ily módon általánosított változatának kidolgozását.

nem általánosítható, viszont más szempontból érdekes: valójában ún. számrendszerváltó program, amely megadja egy adott számnak valamely előírt alapú számrendszerben a számjegyeit. Ilyen algoritmusokat az elektronikus számológépek programozása során állandóan alkalmaznak, de mint a megoldás mutatja, ezeket néha egészen váratlan területeken is fel lehet használni.

**10.** A program:

```
for i:=1 step 1 until n-1 do begin
   for k:=i+1 step 1 until n+1 do a[i, k]:=a[i, k]/a[i, i];
   for j:=i+1 step 1 until n do begin
   T:=a[j, i];
   for k:=i+1 step 1 until n+1 do a[j, k]:=a[j, k]-T×a[i, k]
   end end;
for i:=n step -1 until 1 do begin
   S:=a[i, n+1];
   P : for k:=i+1 step 1 until n do S:=S-a[i, k]×a[k, n+1];
   a[i, n+1]:=S/a[i, i] end   {folytatás}
```

Bizonyos esetekben jól kihasználható a **step** ... **until** ciklusoknak az a tulajdonsága, hogy a ciklus magja egyszer sem kerül végrehajtásra, ha a kezdőérték nagyobb, mint a végérték (pozitív lépésközre gondolunk). Ez a helyzet a *P* címkéjű utasítással, amikor $i=n$.

**11.** A Gauss–Seidel-iteráció programja:

```
for k:=0, k while k<n do begin
   k:=0;
   for i:=1 step 1 until n do begin
      m:=-b[i];  for j:=1 step 1 until n do m:=m+a[j, i]
      ×x[j];
   dx:=-m/a[i, i];  x[i]:=x[i]+dx;  if dx<eps then k:=
   k+1 end end   {folytatás}
```

Látható, hogy az eljárást sokkal könnyebb ALGOL-ban leírni, mint szavakban. Bonyolultabb algoritmusokra ez gyakran előfordul. A *k* változó számlálórekeszként működik; azokat az ismeretleneket számlálja, amelyek javítása már kisebb, mint a pontosságot mérő *epsz* hibakorlát.

**12.** A Gauss–Jordan-mátrixinverzió programja:

```
for i:=1 step 1 until n do begin
   m:=a[i, i];
for j:=1 step 1 until n do begin
   mi:=a[j, i];
   for k:=1 step 1 until n do
      a[j, k]:=if k≠i then a[j, k]-m1×a[i, k]
      else -m1/m end end;   {folytatás}                    (G.5)
```

Úgy hisszük, hogy a korábban tárgyalt és részletesen elemzett programok után ennek a programnak megértése már nem fog különösebb nehézséget okozni. A programban a **then** utáni **for** megengedett variációjára látunk egy konkrét példát.

13. A Horner-módszer programja:

```
b[0]:=a[0];
P:for i:=1 step 1 until n do b[i]:=b[i-1]×x+a[i];
S:=b[0];
for i:=1 step 1 until n-1 do S:=S×x+b[i];
dx:=-b[n]/S;   x:=x+dx;
if abs (dx)>epsz then go to P;   {folytatás}
```

Az $x$-re vonatkozó iterációs ciklust ezúttal nem **for** utasítással szerveztük. A kezdőérték ugyanis itt adva van az $x$ változóban, az új iterált kiszámítása pedig egy bonyolultabb algoritmus végrehajtása során megy végbe, és így a **for** utasítás nyújtotta előnyök jelen esetben nem számottevők.

14. „Bűvös négyzet" előállításának programja:

```
for j:=1 step 1 until n do for k:=1 step 1 until n do
A[j, k]:=0;
j:=1;   k:=n÷2+1;
for i:=1 step 1 until n↑2 do begin
   A[j, k]:=i;   m1:=j+n-1-(j+n-2)÷n×n;
   m2:=k-k÷n×n+1;
   if m1=n∧m2=1∨A[m1, m2]≠0 then j:=j+1 else begin
   j:=m1;   k:=m2 end end;   {folytatás}
```

A program kezdetén zérust írunk a mátrix elemeibe. A mátrixelemek előállítása egy $i$ szerinti ciklusban megy végbe, ahol $i$ az 1-től $n\uparrow 2$-ig terjedő értékeken fut végig. A soron következő mátrixelem indexeinek előállítását speciálisan választott aritmetikai kifejezések végzik. Ezek a II. fejezet 7. gyakorlatból vannak véve. A jobb felső sarokelem leválasztását, valamint annak vizsgálatát, hogy a soron következő elem nincs-e már betöltve, egy külön feltételes utasítás végzi.

## Negyedik fejezet

1. A megfelelő blokkfej:

```
begin real x, y, u, v;   integer i, k, n;
   Boolean ro, pszi;   array A, B[0:6, 1:20];
```

**2. Az azonosítók hatáskörét összefoglaló teljes táblázat:**

| | *B1* | *B2* | *B3* | *B4* |
|---|---|---|---|---|
| *A* | valós lokális | valós lokális | valós globális | valós globális |
| *B* | valós lokális | logikai lokális | logikai globális | logikai globális |
| *J* | valós lokális | valós globális | valós globális | valós globális |
| *K* | valós lokális | valós globális | valós globális | valós globális |
| *i* | egész lokális | egész globális | egész globális | egész globális |
| *j* | nincs deklarálva | egész lokális | egész globális | egész globális |
| *k* | nincs deklarálva | egész lokális | egész globális | egész globális |
| *m* | nincs deklarálva | nincs deklarálva | egész lokális | egész globális |
| *n* | nincs deklarálva | nincs deklarálva | egész lokális | egész globális |
| *p* | egész lokális | egész globális | egész globális | egész lokális |
| *q* | nincs deklarálva | nincs deklarálva | nincs deklarálva | egész lokális |
| *r* | nincs deklarálva | nincs deklarálva | nincs deklarálva | egész lokális |
| *alfa* | nincs deklarálva | logikai lokális | valós lokális | valós globális |
| *beta* | nincs deklarálva | nincs deklarálva | valós lokális | valós globális |
| *gamma* | logikai lokális | logikai globális | logikai lokális | logikai globális |
| *SC* | egész lokális | egész globális | egész globális | egész globális |
| *J6* | egész lokális | logikai lokális | logikai globális | logikai globális |

**3.** Az *n* és *m* egész típusú változók ugyanabban a blokkban vannak deklarálva, mint az *A* tömb, amelynek korlátpárlistájában szerepelnek. Ez ellenkezik a nyelv szabályaival, amint azt a 3. p.-ban láttuk.

**4.** A teljes ügyvitelgépesítési program:

```
begin integer N;  {N beolvasása};
begin integer j, k;  array R, K, B, U, V, Z [1:N];
   integer array I, J [1:N];  real m;
   {Program-vége vizsgálat};
   {Az R, K, B, Z, I és J vektorok beolvasása};
   {A program operatív része};
   {Az R, U és V vektorok kinyomtatása};
   {Visszatérés a program-vége vizsgálatra};  {Megállás} end end
```

Az egyenesillesztési feladatokhoz hasonlóan, a fenti típusú ügyvitel-gépesítési feladatok sem egyediek, hanem általában csoportosan fordulnak elő, ezért a fiktív utasításokkal jelzett vezérprogram tartalmaz egy magasabb rendű ciklust, amelyből akkor lép ki a vezérlés, ha már nincs több feldolgozandó adathalmaz. Ekkor érkezik el a gép a programvégi megálláshoz. Az ügyvitel-gépesítéssel foglalkozó speciális elektronikus számológépek különleges perifériális egységekkel vannak felszerelve, amelyek megkönnyitik a nagy tömegű adat ki- és bevitelt. Ezért a fenti programban a beolvasáson és kinyomtatáson nem feltétlenül szigorúan vett (perforált szalagról vagy lyukkártyáról történő) ki- és bevitel értendő, hanem esetleg az adatoknak az említett speciális perifériális egységről (amely többnyire mágnesszalag) az operativ memóriába és viszont való továbbítása. Ezeknek a feladatoknak a programjait egyébként célszerűbb COBOL-ban irni, mint ALGOL-ban, mert a COBOL-t éppen ilyen problémák algoritmizálására dolgozták ki.

**5.** Primitiv függvény számitásának ALGOL-programja:

```
begin real F, f, m, x;
   F:=0;   f:=1;
   for x:=0 step 0.0001 until 0.4999 do begin
      m:=1/sqrt(1+2×(x+0.0001)+sin(x+0.0001)↑2);
      F:=F+(f+4/sqrt(1+2×(x+0.00005)+sin(x+
      0.00005)↑2+m)×1.66666666₁₀−5;
      f:=m;
{F nyomtatása}
end x;  {megállás} end program
```

6. Egy programozási fogás a ciklus belsejébe való visszaugrásra:

```
begin switch S:=S1, S2;   integer i, N, j, k;
   {utasítások}; j:=1;
   R: for i:=1 step 1 until N do begin
      go to S[j];   S1 : {utasítások};
      B : if L then begin k:=i;  j:=2;  go to P end else go to T;
      S2 : j:=1;  i:=k;  T:{folytatás} end i;
   go to Q;  P:{utasítások};  go to R;  Q:{utasítások} end
```

Az *i* szerinti ciklus közepén a vezérlés valamely *L* logikai feltételtől függően átadódik a cikluson kívüli, *P* címkéjű programrészre. A *P* címkére való elugrás előtt biztosítani kell a ciklusváltozó értékének megőrzését. Erre szolgál a *k* egész változó. A vezérlésnek a program folytatására való visszairányítását az *S* kapcsoló segíti elő. Ennek a kapcsolónak a „normál" állása az *S*1 címke, amikor is $j=1$. A folytatásra (*T* címkéjű programrész) való visszatérés érdekében *j*-t 2-vel tesszük egyenlővé, és az *R* címkére (a ciklus kezdete) ugratunk. Ekkor azonban a ciklus magjának kezdetén elhelyezett vezérlésátadó utasítás helymegjelölő kifejezésének aktuális értéke *S*2 lesz. Az *S*2 címkéjű programrész helyreállítja a ciklusváltozónak a ciklusból való kilépéskor viselt értékét, visszaállítja *S*-t a normál állásba, és folytatódik a program.

7. Vezérlésátadás az *Lpq* címkére:

```
begin switch S: = L00,  L01,  L02,  L03,
                  L10,  L11,  L12,  L13,
                  L20,  L21,  L22,  L23;
integer i, p, q;
   {program, amely értéket ad i-nek};
   p:=i-i÷3×3;  q:=i-i÷4×4;  go to S[4×p+q+1];
   {Utasítások, amelyek tartalmazzák az L00, L01, ..., L23 címkéket}
   end
```

A kapcsolókifejezések tárgyalásakor utaltunk az indexes változókkal való formai és elvi analógiára. Most rámutatunk arra, hogy az említett analógia az egyindexes változókra vonatkozik. Többindexes változóknak nincs megfelelője a kapcsolókifejezések körében. Az *S* kapcsoló listaelemeit az áttekinthetőség kedvéért egy $3\times4$ típusú mátrixba rendeztük. Így az *Lpq* címke e mátrix $p+1$-edik sorának $q+1$-edik eleme lett ($p=0, 1, 2$ és $q=0, 1, 2$ és 3). Ez az elrendezés azt sugalmazza, hogy a kívánt vezérlésátadást végezzük egy **go to** $S[p+1, q+1]$ alakú utasí-

tás segítségével. Csakhogy ilyen utasítást az ALGOL szabályai nem engednek meg. Az a tény, hogy a kapcsolólista elemeit mátrixba rendeztük, mit sem változtat a kapcsolódeklaráció szemantikáján. Ezért a kívánt kapcsolólista-elem megkeresésére egy alkalmas aritmetikai kifejezést kellett konstruálnunk. Ilyen „sokcímkés" elágazás valamely címkemátrix elemeire az ALGOL szokványos alkalmazási területein nem tipikus, ezért a többindexes kapcsolókifejezések hiánya nem komoly megszorítás. Annál is kevésbé, mert mint láttuk, aránylag könnyen megkerülhető.

8. Teljes program két mátrix szorzatának meghatározására:

```
begin integer n, m, r;
   {n, m és r beolvasása};
   begin array A [1:n, 1:m], B [1:m, 1:r], C [1:n, 1:r];
      integer i, j, k;   real S;
      {A és B beolvasása};
      for i:=1 step 1 until n do for j:=1 step 1 until r do begin
         S:=0;
         for k:=1 step 1 until m do S:=S+A [i, k]×B[k, j];
         C [i, j]:=S end j és i;
   {A C mátrix kinyomtatása}; {Megállás} end end
```

9. A feladatnak bonyolult feltételes utasítások segítségével való megoldása helyett, célravezetőbb egy kapcsoló alkalmazása. A programséma a következő lehet:

```
begin Boolean array B [1:3]; integer i, j, k;
   switch L:=S3, S1, S2, S2, S1, S3, S1, S2;
   {utasítások, amelyek értéket adnak a B vektor elemeinek};
   i:=if B [1] then 4 else 0;   j:= if B [2] then 2 else 0;
   k:= if B [3] then 1 else 0; P: go to L [i+j+k+1];
   {utasítások, amelyek tartalmazzák az S1, S2 és S3 címkéket} end
```

Ez a program azt is mutatja, hogy egy kapcsolólista elemeinek nem kell szükségképpen különbözőknek lenniük. Az *L* kapcsoló listája például 8 elemből áll, elemei között viszont csak három különböző található, ti. *S*1, *S*2 és *S*3.

10. Egyikre sem. A közölt ALGOL-program ugyanis szintaktikusan hibás: A *B* címkéjű utasítás nem blokk, hanem összetett utasítás. Ilyenformán ugyanaz az *L* azonosító kétszer fordul elő ugyanabban a blokkban.

Más lenne a helyzet, ha *B* **begin**-je után legalább egy deklaráció állna. Ebben az esetben *B* is blokk lenne, és ily módon szintaktikusan

kifogástalanná válna a program. A **go to** utasítás az ismert szemantikai szabályok értelmében az *A* blokkbeli *L* címkére vezetne.

## Ötödik fejezet

1.

```
procedure MVMULT(A, v, x, n); value n;
array A, v, x; integer n;
  begin real S; integer i, j;
    for j:=1 step 1 until n do begin
      S:=0; for i:=1 step 1 until n do S:=S+A[j, i]×v[i];
      x[j]:=S end j end MVMULT
```

2.

```
procedure LINFIT(x, y, A, B, C, Q, N); value N;
integer N;
  array x, y; real A, B, C, Q;
  begin real S1, S2, S3, S4, S5, t, u, D; integer i;
    {A 3.76 program törzse} end LINFIT
```

3. A gyakorlatnak megfelelő teljes program a következő lehet:

```
begin array A[1:9]; integer i; real t;
  real procedure F(x); value x; real x; F:=sin(1/x);
  procedure MULLER (f, epsz, z1, z2, z3, root); value epsz, z1, z2, z3;
    real procedure f; real epsz, z1, z2, z3, root;
    begin real f1, f2, f3, h3, u, v, w, lambda, delta, nev, g;
    integer i;
    f1:=f(z1); f2:=f(z2); h3:=z3-z2; lambda:=h3/
    (z2-z1);
    for i:=3, i+1 while abs (h3/z3)≧epsz do begin
      delta:=1+lambda; f3:=f(z3); u:=f3×delta;
      v:=f1×lambda; w:=f2×delta; g:=u+lambda×(v+
      f3)-w×delta;
      nev:=sqrt (g↑2-4×u×lambda×(v-w+f3));
      v:=g+nev; w:=g-nev;
      nev:=if abs (v)>abs (w) then v else w; lambda:=-
      2×u/nev;
      h3:=lambda×h3; f1:=f2; f2:=f3; z1:=z2; z2:=z3;
      z3:=z3+h3 end i; root:=z3 end MULLER;
```

```
comment Itt kezdődik a tulajdonképpeni program:

A [1]:=0.25;  A [2]:=0.35;  A[3]:=0.45;
A [4]:=0.13;  A[5]:=0.17;
A [6]:=0.205;  A [7]:=0.0905;  A [8]:=0.1;
A [9]:=0.12;
for i:=1 step 3 until 7 do begin
    MULLER (F, 10-6, A [i], A [i+1], A [i+2], t);
    {t és esetleg a kezdőértékek kinyomtatása} end i end program;
```

*Megjegyzés:* a fenti programot gyakorlatban is kipróbáltuk, és az iteráció általában 4—5 lépésben beállította a gyök pontos értékét.

4. A legnagyobb közös osztó meghatározásának ALGOL-programja a következő:

```
integer procedure LNKO (p, q);  value p, q;  integer p, q;
  begin integer r;
    for r:=q while q>0 do begin q:=p-p÷q×q;  p:=r
    end r;
  LNKO:=p end LNKO,
```

5. Gyakorlati szempontok miatt az egyváltozós szélsőérték-kereső eljárást egy kissé másként célszerű megírni, ha azt önálló, „közérdekű" eljárásként akarjuk kiképezni, mint ha az az (5.34) eljárásdeklaráció mintájára, a kétdimenziós eljárás törzsének egy lokális eljárása kíván csupán lenni. Ezért az egyváltozós eljárást bemutatjuk külön is, és a kétdimenziós szélsőérték-keresés céljaira kissé átalakítjuk majd.

*a)* Eljárásdeklaráció egyváltozós függvény szélsőértékének meghatározására:

```
procedure MAX (f, M, Mx, x0, delta x, epsz, eta):  value x0, delta x,
epsz, eta;
  real M, x0, delta x, epsz, Mx; real procedure f;
  begin real m, me, fu, fe, x;
    x:=x0;  fe:=f(x);  m:=0;  me:=epsz;
    for x:=x+delta x while abs(m-me)≧eta ∨ abs(delta x)≧epsz do
    begin
      fu:=f(x);
      if fu<fe then begin delta x:=-delta x/3;  me:=m; m:=fe;
      Mx:=x-delta x end;
    fe:=fu end x;  M:=m end MAX
```

*b)* Eljárásdeklaráció kétváltozós függvény szélsőértékének meghatározására:

```
procedure MAX2 (F, K, A, S, delta, epsz, eta); value delta,
epsz, eta;
   real delta, epsz, eta, S; real procedure F; array K, A;
   begin real XX, YY, MX, MY, MAXM, MAXME, MXE,
    MYE;
      procedure MAX1 (f, M, mx, x0, x); value x0; real f,
      M, mx, x0, x;
         begin real m, me, fu, fe, delta x;
            x:=x0; fe:=f; m:=0; me:=epsz; delta x:=
            delta;
            for x:=x+delta x while abs (me-m)≧
            eta ∨ abs (delta x)≧epsz do begin
            fu:=f; if fu<fe then begin delta x:=-delta x/3;
            me:=m; mx:=x-delta x; m:=fe; end
            fe:=fu end x; M:=m end MAX1;
comment Itt kezdődik MAX2 eljárástörzse;
MXE:=MX:=K [1]; MYE:=MY:=K [2];
MAXM:=0;
MAXME:=eta;
L:if abs (MAXM-MAXME)≧eta ∨ sqrt ((MXE-
   MX)↑2+(MYE-MY)↑2)≧epsz then begin
   MAXME:=MAXM; MXE:=MX; MYE:=MY;
   MAX1 (F(XX, MY), MAXM, MX, MX, XX);
   MAX1 (F(MX, YY), MAXM, MY, YY); go to L
   end;
A [1]:=MX; A [2]:=MY; S:=MAXM end MAX2
```

A *MAX*-eljárás átdolgozása főként abból állt, hogy alkalmassá kellett tenni kétváltozós függvényen való operálásra is. *MAX* erre nem képes, mivel az eljárástörzsben a formális paraméterként bevezetett *f egyváltozós* függvény. Ezt a nehézséget úgy kerültük meg, hogy *MAX*1-ben *f*-et **real procedure** helyett **real**-ként specifikáltuk, és ennek megfelelően az aktuális paraméterként fellépő *F* függvényt aktuális argumentumaival együtt adtuk meg az eljárást aktivizáló eljárásutasításokban, hogy azok korrekt aritmetikai kifejezések legyenek. Egy függvényeljárás azonosítója aktuális argumentumaival együtt ugyanis korrekt aritmetikai kifejezés, de argumentumok nélkül nem az (kivéve, ha az őt meghatározó eljárásdeklarációnak nincs formális paramétere). Az egyéb változtatások nem lényegesek: mindössze néhány formális paramétert (*epsz*, *eta*, *delta*) globálissá tettünk az eljárástörzsre nézve.

6. Eljárásdeklaráció „önosztó" típusú Simpson-integrációra:

```
procedure SIMPSON (a, b, f, epsz, NO, INT); value a, b,
epsz, NO;
real a, b, epsz, INT; integer NO; real procedure f;
  begin real S1, S2, I, IE, v, h, delta, i: integer j, k, N;
    switch Z:=Z[j], Z1, Z2;
  Bejárat: k:=1; delta:=IE:=S2:=0; N:=NO; j:=3;
  Repeat: S1:=0; h:=(b-a)/N;
    for i:=delta step 1 until N do begin
      v:=f(a+i×h); if i=0∨i=N then v:=v/2;
      go to Z[k]; Z1: S1:=S1+v; j:=3; go to L;
      Z2: S2:=S2+v; j:=2; L: end i;
      I:=(4×S1+2×S2)×h/3/k;
      if k=1 then begin k:=2; delta:=0.5 end else N:=2×N;
    if abs ((I-IE)/I)≧epsz then begin S2:=S1+S2; IE:=I;
      go to Repeat end
      else INT:=I+1.0666666666×(I-IE) end SIMPSON
```

Azt hisszük, a programhoz nem kell sok kommentárt fűznünk; annyit talán megjegyzünk, hogy az Olvasónak érdemes megfigyelnie a *Z* kapcsoló működését, amely biztosítja, hogy a kezdeti felosztás mellett kiszámításra kerüljön mind az $S_1$, mind az $S_2$ szumma, azonban a későbbiekben már csak $S_1$ (hiszen $S_2$ az előző részeredmények felhasználásából adódik).

7. *a)* Rekurzív eljárás polinom helyettesítési értékének meghatározására:

```
real procedure HORNER (x, n, A); value x; real x;
integer n; array A;
  HORNER:=if n=0 then A [0] else HORNER (x, n-1, A)
  × x+A [n];
```

*b)* Rekurziv eljárás számsorozat *n*-edik elemének meghatározására:

```
real procedure a(n); value n; integer n;
  a:=if n=0 then 2 else if n=1 then 5 else 1/a(n-2)+
  1/a(n-1);
```

*c)* Rekurzív eljárás *n*-szeres integrál numerikus kiszámítására:

```
real procedure TI (a, b, n, f, h); value a, b, n, h;
   array a, b: integer n; real h: real procedure f;
comment Az a tömb elemei tartalmazzák az integrációs tartomány
alsó, a b tömb elemei pedig az integrációs tartomány felső határait;
   begin real procedure EI(p, q, F, y); value p, q; real p, q,
   F, y;
      begin real S; S:=0;
      for y:=p step h until q do S:=S+F; EI:=h×S
      end EI;
   array x [1:n];
   TI:= if n=1 then EI (a[1], b[1], f(x), x[1])
   else EI (a[n], b[n], TI(a, b, n-1, f, h), x[n]) end TI
```

Az előző példákkal azt kívántuk illusztrálni, hogy bizonyos esetekben még numerikus jellegű feladatokban is alkalmazhatók rekurzív eljárások. Különösen érdekes ebből a szempontból a *c)* példa, melynek direkt algoritmussá való átfogalmazása egyáltalán nem magától értetődő feladat. Az olyan eljárásokban, amelyek *n*-változós függvényeken operálnak, ahol *n* maga is paraméter, gyakran tapasztaljuk, hogy egyes algoritmusokat sokkal kényelmesebben lehet rekurzíve, mint direkt módon megfogalmazni. Sajnos a rekurzív algoritmusok gépidő szempontjából általában nagyon is gazdaságtalanok, és ezért a programozó inkább vállalja a direkt algoritmussá való átfogalmazással járó fejtörést.

8. Az (5.55) program blokkfejében három eljárás van deklarálva. A *COLLECT* eljárást nevezhetnénk „összegyűjtő" eljárásnak, amely meghatározza azokat az elemeket, amelyeken egy meghatározatlan (formális paraméterrel kijelölt) eljárás operál. *COLLECT* törzsében ez a meghatározatlan „operatív" eljárás az *OPERATIV* azonosítóval van ellátva. Hogy ténylegesen mi hajtódik végre a *COLLECT* által „összegyűjtött" elemeken, az akkor dől el, amikor *COLLECT* aktivizálásra kerül, és megadjuk számára az aktuális „végrehajtó" eljárás azonosítóját.

A *COLLECT* eljárásnak pontos ellentéte a *DO* eljárás, amely éppen „végrehajtó" jellegű. Ez egy meghatározatlan (formális paraméterrel kijelölt) elemen egy meghatározott akciót hajt végre, ti. zérussal teszi egyenlővé. Az, hogy miféle elemet kell zérussal egyenlővé tennie, csak az eljárás aktivizálásakor dől el, ti. amikor *S* helyébe beírjuk az aktuális argumentum azonosítóját.

A blokkfejben deklarált harmadik eljárás, a *GO*, „aktivizáló" funkciót tölt be. Mindössze annyit tesz, hogy behív egy határozatlan „összegyűjtő" eljárást, egy ugyancsak határozatlan „végrehajtó" eljárással, mint aktuális paraméterrel. Az, hogy mely összegyűjtő eljá-

rást hívja be, és milyen aktuális végrehajtó eljárást rendel hozzá, *GO* aktuális paramétereiből derül ki. Adott esetben a *COLLECT* összegyűjtő eljárást hívja be *DO* aktuális argumentummal.

A fentiek szerint a felírt program zérust ír a *P* vektor valamennyi elemébe. Ugyanis az egymáshoz kapcsolódó eljárásbehívások láncolata végeredményben a *DO* eljárást aktivizálja minden egyes *P*[*i*] vektorelemre, mint aktuális argumentumra.

Ha a programot úgy akarjuk átalakítani, hogy az ne nullát írjon *P* minden egyes elemébe, hanem pl. megkeresse *P* maximális abszolút értékű elemét, akkor nyilván a végrehajtó eljáráson kell változtatnunk; az összegyűjtőn nem, hiszen operálni ugyanazokon az elemeken kell, mint az előbb, csak maga az operáció más. A megfelelő átalakítás a következő lehet: feltételezzük, hogy *maxP* szintén globális azonosítója a blokknak, és oda kívánjuk a maximális abszolút értékű elemet beírni. A programban az első két eljárásdeklaráció változatlan, csak a *DO* eljárást kell kicserélni, és a blokk törzsén egy apró változtatást eszközölni (csak a program végét írjuk le újra):

```
procedure DO1(S); real S;
  if abs(S) > abs(maxP) then maxP := S;
  maxP := 0; GO(DO1, COLLECT) end
```

A *b)* feladat szerint a blokkot úgy kell átdolgozni, hogy az három másik globális változót tegyen egyenlővé zérussal. Ebben az esetben éppen az összegyűjtő eljárást kell kicserélnünk, és minden egyéb változatlan marad. Az új összegyűjtő eljárás a következő lehet:

```
procedure COLLECT1 (OPERATIV); procedure
OPERATÍV;
  begin OPERATIV(A); OPERATIV(B); OPERATIV(C)
  end COLLECT1;
```

A *GO* eljárás behívása most a következő lesz:

*GO* (*DO*, *COLLECT*1);

Megjegyezzük még, hogy a következő blokktörzs:

*maxP* := 0; *GO* (*DO*1, *COLLECT*1)

az *A*, *B*, *C* számok abszolút értékben legnagyobbikának értékét helyezi el a *maxP* változóba.

Bár a fentiek látszólag játékos ügyeskedésnek tűnhetnek, meg kell jegyeznünk, hogy ilyen „összegyűjtő" és „végrehajtó" eljárásoknak komoly szerepük lehet pl. fordítóprogramok logikai blokkjaiban.

# AZ ALGOL 60 HIVATKOZÁSI NYELV ALAPJELEI

Betűk: *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*, *K*, *L*, *M*,
*N*, *O*, *P*, *R*, *S*, *T*, *U*, *V*, *Z*, *X*, *Y*, *W*, *Q*,
*a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l*, *m*,
*n*, *o*, *p*, *r*, *s*, *t*, *u*, *v*, *z*, *x*, *y*, *w*, *q*

Számjegyek: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Aritmetikai műveleti jelek: +, −, ×, /, ÷, ↑

Logikai műveleti jelek: ∨, ∧, ⊃, ≡, ¬

Relációjelek: >, ≧, =, <, ≦, ≠

Zárójelek: (, ), [, ]

Írásjelek: . (pont), , (vesszõ), :, ;, ', '

Hatványalap jele: $_{10}$

Értékadás jele: :=

Szóköz jele: ␣

Szó-alapjelek:

| | | | |
|---|---|---|---|
| **array** | = elrendezés (tömb) | **label** | = címke |
| **begin** | = kezdet | **own** | = saját |
| **Boolean** | = Boole-féle | **procedure** | = eljárás |
| **comment** | = magyarázat | **real** | = valós |
| **do** | = tedd | **step** | = lépés |
| **else** | = egyébként | **string** | = jelsorozat, idézet |
| **end** | = vég | **switch** | = kapcsoló |
| **false** | = hamis | **then** | = akkor |
| **for** | = részére, -ra, re | **true** | = igaz |
| **go to** | = menj ... hez | **until** | = -ig |
| **if** | = ha | **value** | = érték |
| **integer** | = egész | **while** | = mindaddig, amíg |

Ha az ábécé betűinek a „Betűk" rovatban felsorolt szimbólumokat választjuk, akkor az alapjelek száma 116, ebből

| | |
|---|---|
| betű | 52, |
| számjegy | 10, |
| alapszó | 24, |
| egyéb jel | 30. |
| összesen: | 116 |

Az egyéb jelek közé beszámítottuk a ␣ jelet is, amelynek a hivatkozási nyelvben tulajdonképpen nincs szerepe.

# IRODALOM

## *A)* Bevezető munkák


[1] *Gnyegyenko, B. V. és munkatársai:* Bevezetés a programozásba. I—II. Műszaki Könyvkiadó, Budapest, 1964.

[2] *Kitov, A. I. — Krinyickij, N. A.:* Elektronikus digitális számológépek és programozás. Akadémiai Kiadó, Budapest, 1964.

[3] *Cracken, D. Mc.:* A guide to ALGOL programming. Wiley, 1962.

[4] *Bottenbruch, H.:* Structure and Use of ALGOL 60. Journal of the ACM, 1962. 9. köt. 2. sz. 161. old.

[5] *Dijkstra, E. W.:* A Primer to ALGOL 60 programming. Academic Press, 1962.

[6] *Vilstrup, H.:* An ALGOL Training Manual. Regnecentralen, Copenhagen, 1963.

[7] *Higman, B.:* What EVERYBODY should know about ALGOL. The Computer Journal, 1963. 6. köt. 50. old.

[8] *Cracken, D. Mc.:* A guide to FORTRAN programming. Wiley, 1961.

[9] *Cracken, D. Mc.:* A guide to COBOL programming. Wiley, 1963.


## *B)* Egyéb irodalom


[10] *Naur, P.* [szerk.]: Report on the Algorithmic language ALGOL 60.
*a)* Communications of the ACM*, 1960. 3. köt. 2. sz. 299. old.
*b)* Numerische Mathematik, 1963. 2. köt. 106. old.

[11] *Naur, P.* [szerk.]: Revised Report on the Algorithmic Language ALGOL 60.
*a)* CACM, 1963. 6. köt. 1. old.
*b)* Numerische Mathematik, 1963. 4. köt. 420. old.

[12] *Naur, P.* [szerk.]: Jelentés az ALGOL 60 algoritmikus nyelvről. [10] magyar nyelvű fordítása. MTA Matematikai Kutató Intézetének Közleményei, *B* sorozat, 1961. 6. köt. 4. sz. 425. old.


* A továbbiakban CACM.

[13] *Naur, P.* [szerk.]: Módosított jelentés az ALGOL 60 algoritmikus nyelvről. [11] magyar nyelvű fordítása. „Az ALGOL 60 nemzetközi algoritmikus nyelv" a NIMDOK és a NIM Ipargazdasági és Üzemszervezési Intézetének közös kiadványa, 1965.
[14] *Perlis, A. J.—Samelson, K.* [szerk.]: Preliminary Report — International Algebraic Language. CACM, 1958. 1. köt. 12. sz.
[15] *Perlis, A. J. — Samelson, K.* [szerk.]: Report on the Algorithmic Language ALGOL by the ACM Commitee on Programming Languages and the GAMM Commitee on Programming. Numerische Mathematik, 1959. 1. köt. 41. old.
[16] *Backus, J.*: The syntax and semantics of the proposed international algebraic language of the Zürich ACM-GAMM conference. ICIP, Paris, June, 1959.
[17] *Van der Poel, W. L.* [szerk.]: Report on Input — Output Procedures for ALGOL 60. Numerische Mathematik, 1964. 6. köt. 5. sz. 459. old.
[18] *Gorn, S.* [szerk.]: A Proposal for Input—Output Conventions in ALGOL 60. CACM, 1964. 7. köt. 5. sz. 273. old.
[19] *Bachelor, G. A. és mások*: SMALGOL 61. CACM, 1961. 4. köt. 11. sz. 499. old.
[20] *Van der Poel, W. L.*: Report on SUBSET ALGOL 60. Numerische Mathematik, 1964. 6. köt. 5. sz. 454. old.
[21] *Samelson, K. — Bauer, F. L.*: Sequential Formula Translation. CACM, 1960. 3. köt. 2. sz. 76. old.
[22] *Grau, A. A.*: Recursive processes and ALGOL translation. CACM, 1961. 4. köt. 1. sz. 10. old.
[23] *Woodger, M.* [szerk.]: Supplement to the ALGOL 60. Report CACM, 1963. 6. köt. 1. sz. 18. old.
[24] *Gorn, S.* [szerk.] *és munkatársai:* A Programming Language for Information Processing on Automatic Data Processing Systems. FORTRAN vs. Basic FORTRAN. CACM, 1964. 7. köt. 10. sz. 590. old.

# TARTALOM


Előszó ..... 5

Első fejezet: Az elektronikus számológépek programozásának automatizálása és az ALGOL 60 programozási nyelv ..... 7

1. Bevezetés ..... 7
2. Az ALGOL 60 nyelv célja, szintjei ..... 10
3. Alapfogalmak ..... 11

Második fejezet: Aritmetikai és logikai kifejezések ..... 17

1. Bevezetés ..... 17
2. Számok ..... 18
3. Betűk, változók és azonosítók ..... 20
4. Aritmetikai műveletek ..... 23
5. Feltétlen aritmetikai kifejezések ..... 26
6. Logikai változók és logikai műveletek ..... 37
7. Relációk ..... 40
8. Feltétlen logikai kifejezések ..... 42
9. Gyakorlatok ..... 50

Harmadik fejezet: Utasítások ..... 54

1. Bevezetés ..... 54
2. Értékadó utasítások ..... 54
3. Címkék és vezérlésátadó utasítások ..... 60
4. Feltételes utasítások és feltételes kifejezések ..... 62
5. Összetett utasítások ..... 77
6. Indexes változók ..... 80
7. Ciklusutasítások ..... 86
   *a)* A ciklusutasítás első alaptípusa ..... 88
   *b)* A ciklusutasítás második alaptípusa ..... 91
   *c)* A ciklusutasítás harmadik alaptípusa ..... 94
   *d)* Az általános típusú ciklusutasítás ..... 96
8. Programozási példák ..... 103
9. Gyakorlatok ..... 116

Negyedik fejezet: A program kísérő információi ..... 123

1. Bevezetés ..... 123
2. Típusdeklarációk ..... 125
3. Tömbdeklarációk ..... 131
4. Kapcsolók, helymegjelölő kifejezések ..... 139
5. A címkék hatásköre ..... 149
6. Magyarázó megjegyzések a programban ..... 153
7. Gyakorlatok ..... 155

Ötödik fejezet: Eljárások ..... 159

1. Bevezetés ..... 159
2. Eljárásdeklaráció és eljárásutasítás. Aktuális és formális paraméterek ..... 161
3. Példák eljárások alkalmazására. A specifikációs rész ..... 172
4. Függvényeljárások ..... 190
5. Rekurzív eljárások és rekurzív eljárásbehívások ..... 202
6. Paraméterelválasztók és idézetek ..... 209
7. A perifériális egységekre hivatkozó utasítások ..... 212
8. Gyakorlatok ..... 219

Függelék. A metanyelvi leírás módszere ..... 225

A gyakorlatok megoldásai ..... 229

Az ALGOL 60 hivatkozási nyelv alapjelei ..... 249

Irodalom ..... 251

Műszaki vezető: Hegedűs Ernő — Műszaki szerkesztő: Bacsó Katalin
Borító- és kötésterv: Horváth Attila
A könyv formátuma: Fr5 — Ívterjedelem: 12,75 (A5)
Ábrák száma: 6 db — Példányszám: 7800
Papír minősége: 80 g ofszet — Betűcsalád és -méret: Times bg/gm
Azonossági szám: 60 890 — Mű: 2575 — i — 7881
Készült az MSZ 5601 59 és 5602—67 szerint
Szedte a Nyomdaipari Fényszedő Üzem (779607/9)
78-64 Sylvester János Nyomda, Szombathely

Ára: 19,50 Ft

ALGOL 60